TECHNISCHE UNIVERSITÄT
CHEMNITZ

Generating Formal Representations of System Specification from Natural Language Requirements

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Zeeshan Irfan
Student ID: 460004
Date: 11.07.2019

Supervising tutor: Prof. Dr. W. Hardt
Mr. Christian Amey (Bertrandt Ingenieurbüro GmbH)
Mr. René Bergelt (Internal Supervisor)

Abstract

Natural Language (NL) requirements play a significant role in specifying the system design, implementation and testing processes. Nevertheless, NL requirements are generally syntactically ambiguous and semantically inconsistent. Issues with NL requirements can result into inaccurate and preposterous system design, implementation and testing. Moreover, informal nature of NL is a major hurdle in machine processing of system requirements specifications. To confront this problem, a requirement template is introduced, based on controlled NL to produce deterministic and consistent representation of the system. The ultimate focus of this thesis is to generate test cases from system specifications driven from requirements communicated in natural language. Manual software systems testing is a labour intensive, error prone and high cost activity. Traditionally, model-driven test generation approaches are employed for automated testing. However, system models are created manually for test generation. The test cases generated from system models are not generally deterministic and traceable with individual requirements. This thesis proposes an approach for software system testing based on template-driven requirements. This systematic approach is applied on the requirements elicited from system stakeholders. For this purpose natural language processing (NLP) methods are used. Using NLP approaches, useful information is extracted from controlled NL requirements and afterwards the gathered information is processed to generate test scenarios. Our inceptive observation exhibits that this method provides remarkable gains in terms of reducing the cost, time and complexity of requirements based testing.

Keywords: Natural Language, Requirements templates, NLP

Contents

Acknowledgement

I am grateful to Mr. Christian Amey for providing me an opportunity to work in his group, and being my mentor for his guidance throughout this thesis work. I also would like to thank Bertrandt Ingenieurbüro GmbH for the financial support provided during the whole project. Specially grateful to Prof. Dr. Wolfram Hardt for granting great supervision under his chair and internal supervisor Mr. René Bergelt for his valuable input and advise time to time. The author is really thankful to the members of Electric/Electronic department at Bertrandt for their involvement and feedback all through this research work. .

List of Figures

List of Tables

List of Abbreviations

NL Natural Language

NLP Natural Language Processing

UML Unified Modeling language

sysML Systems Modeling Language

SDL Specification Description Language

EFSM Extended Finite State Machine

UMGAR UML Model Generator from analysis of Requirements

SBVR Semantic of Business Vocabulary and Rule

RE Requirement Engineering

SDLC Software Development Life Cycle

TBD To Be Determined

SRS Software Requirement Specification

SR System Requirement

UR User Requirement

HMI Human Machine Interface

POS Parts of Speech

ER Entity Relation

NLTK Natural Language Toolkit

GUI Graphical User Interface

ECU Electronic Control Unit

ECT Engine Coolant Temperature

ECP Equivalence Class Partitioning

SC Structural Complexity

1. Introduction

According to an estimation approximately 87.7% of system specification documents are written in natural language (NL), while 5.4% of the documents are stated in formal language, and remaining are expressed using other methods [1]. Therefore, the most utilized techniques to record requirements is the use of NL such as English, since requirements in NL form are easy to specify, verify, employ and are generally accepted. However, NL requirements are fundamentally ambiguous, inconsistent and incomplete in nature [2]. In order to construct software models, impermeable and clear software requirements are essentially required as machine cannot process accurately syntactically ambiguous requirements. It is usually the task of requirement analyst to uncover and resolve possible ambiguities, inconsistencies and incompleteness in stakeholders requirements [3]. To overcome this problem, different techniques are defined in terms of documenting the requirements and a requirement template is proposed to extract controlled NL. This thesis aims at requirement based software testing using natural language processing. Considering the aspect that software testing plays a major role in the software systems verification and it imposes improvements in the development of system. Software development for embedded systems is fundamentally different due to the restricted interaction possibilities with the system, e.g. for debugging purposes. Therefore, the testing phase of both software and hardware is essential in system development to make sure requirements are met. This is even more important in safety-critical environments like automotive, railroad, aerospace or medical applications.

There exist certain approaches to generate test cases with the help of semi-formal or formal representation of system requirements such as use case, activity diagrams and sequence diagrams. These approaches have their own restrictions as they lack the ability to seize test cases for non functional requirements [4]. In order to formalize the system specifications more expertise is required, hence it is time consuming and difficult for the testers and other domain specialists to test the system. Formalization of system specifications has a high cost which many companies are hesitated to pay for [5]. Test case generation from formal representations is considered as one of the difficult step in software development life cycle. Software testing takes up to 40 to 60 percent of total effort, while the cost of correction is even higher [6]. Manual software testing for embedded systems is labour intensive and error prone, therefore influences the effectiveness and efficiency of software [7], [8].

1.1. Motivation

Manual embedded software testing is used in various security critical environments such as automotive, railroad, aerospace or medical applications. For the verification and validation of such systems, the test cases must be deterministic and traceable with the requirements to prevent unwanted consequences which can threatened Human safety.

This thesis focuses on the formalization of informal user requirements in a format that helps in the easy automation of test cases [9]. The formal representation of system specifications from natural language requirements result in less cost and complexity. There are certain benefits associated with early testing of system or design of test cases in the requirement stage in the following ways [10].

1. It assists system designer and analyst to express and know the requirements in a better way.
2. Misunderstanding of the requirements is confronted.
3. Requirements testability is made certain

1.2. Problem Statement

One of the possible source for defects in manual software testing is the human factor. Especially, when it comes to repeating complex tasks, the possibility of man-made mistakes is ever-present. One approach to eliminate human error is the use of automated test case generation approaches. (Getter/Setter in IDE, Vector AUTOSAR components [11]). Such code generators produce deterministic results and can be qualified by means of functional safety.

Problems associated with automated generation of test cases from NL requirements specification are that NL requirements are syntactically ambiguous and semantically inconsistent [12], therefore result into non-deterministic and non traceable tests cases. Furthermore, formal modeling techniques are used for the generation of test cases such as (UML, sysML etc.). Manually creating formal representation of system specifications from NL requirements require domain knowledge, therefore costly, time consuming and a complexity activity. Other than that, each developer has his own interpretation of the system thus formal models created by domain experts might remain non deterministic.

In order to overcome these problems a systematic approach is applied, in which NL requirement statements are represented in control NL using requirement template. The formal representation of system specifications are then processed by NLP methods for the generation of test scenarios. The technical overview of the proposed approach is illustrated in Figure 1.1. Objectives of this research are as follows:

- To specify the current state of affairs and challenges related to automated test case generation using the natural language requirement.

- Bridge the gap between the informal textual requirements and formal models, thus test cases can be generated based on template driven requirements.
- To find out if the problem of deterministic and traceable test cases solvable by implementing requirement templates.

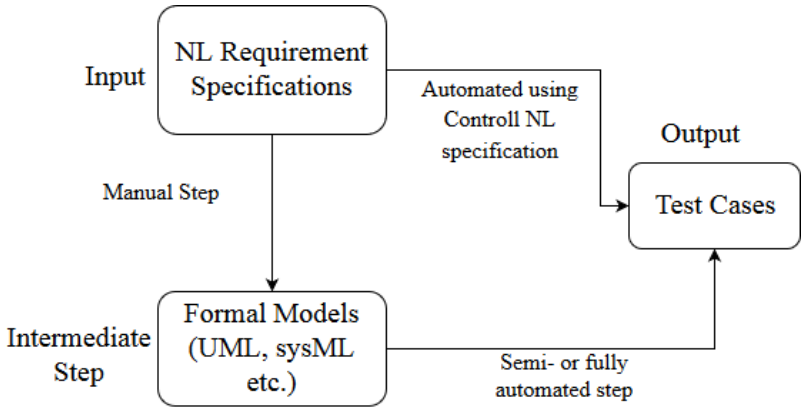

Figure 1.1.: Software testing technical overview

1.3. Thesis Structure

This thesis is structured as follows:

- Basics: This chapter elaborates the problems associated with natural language requirements specifications, its formal representation, and also discusses about why manual software testing is an issue.
- State of the Art: In this chapter, relevant approaches in perspective to this thesis are discussed.
- Concept: This chapter provides a systematic approach to solve the addressed problem based on the guidelines discussed in the chapter 3. Furthermore, this step-by-step approach provides appropriate examples at each level for better understanding.
- Formal Requirement Criteria: In this chapter, number of instructions are provided in order to write formal requirements. Furthermore, attributes of a well written requirement statement are discussed. In the end, requirement style elements are explored in detail.
- Implementation: This chapter provides an overview of tools used during the implementation phase. Furthermore, a detail description about the implementation of proposed solution is also provided.

- Evaluation: This chapter discusses the defined goals and evaluate how far these goals are accomplished. Additionally, results obtained after the implementation phase and issues observed during the implementation stage are analyzed.
- Conclusion: This chapter provides a brief summary of this thesis. It also expresses the future work which can improve the suggested solution.

2. Basics

2.1. Manual Software Testing Problems

A major part of test cases are written manually, making this critical phase a time consuming, and a laborious intensive activity. In manual software testing it is rare to achieve tolerable coverage and it is not repeatable. Therefore, it affects the efficiency and effectiveness of the software.
One of the potential problem is the human involvement. Especially when it comes to repeating complex tasks, the possibility of common human mistakes are ever-present, which may result into unwanted consequences. Examples are STS-51-L [13] or ARIANE V88 [14]. One approach to eradicate human error is the use of code generators, e.g Getter/Setter in IDE, Vector AUTOSAR components. Such code generators produce deterministic results and can be qualified by means of functional safety.

2.2. Why NL Requirements?

Traditionally, software requirements specifications are expressed in natural languages. From the recent estimation, around 87.7% of requirements documents are stated in natural language while 5.3% of the requirements specifications are represented in formal languages and remaining are illustrated using other methods [15]. The requirements are documented in natural languages such as English because such representation of requirements are easier to state, understand, verify and are accepted universally by humans. These are the major reasons why requirements are specified in natural language despite the fact that NL requirements can be syntactically ambiguous, incomplete and semantically inconsistent [12]. The human analyst may fail to observe problems in NL requirements which can result in more than one interpretation for the requirement due to the lack of domain knowledge. The significance of NL requirements will be addressed in detail later in this thesis. Following are different forms of NL requirements:

- NL Statements, obeying or not with templates. [16]
- User Stories, complying different templates.
- Use case specification, can be structured and regulated. [17]
- Combination of NL and models (e.g. activity diagram). [18]

2.3. Requirements Formalization Issues

Formal representation of system specifications can eradicate ambiguity using methods such as UML statecharts, finite state machine and some other methods [19]. However, they do not guarantee to resolve other issues with requirement engineering (RE) e.g. UML is unable to express the non functional specifications of the system. These techniques have limitations in acceptance testing and are not successful for the large scale software system verification [20].

When the requirements are represented with the help of semi-formal or formal methods then it will be hard for the software tester to test the system as domain knowledge is needed for the interpretation of requirements and only few domain experts could understand that [4]. Furthermore, formal modelling of requirement specification is costly and a time consuming endeavour.

2.4. Requirement Engineering Introduction

In the software development life cycle (SDLC), requirement engineering (RE) is a crucial and an early stage. There are four important tasks in the RE stage such as 1) requirement elicitation, 2) documentation of requirements, 3) testing and 4) validation. This systematic method helps the team of developers to sufficiently communicate with system stakeholders. The gathered requirement statements are then properly written into formal requirement specification document [21].

2.4.1. NL in Requirement Elicitation

The statements of stakeholder are usually expressed in NL during elicitation of requirements. The use of NL in expressing the requirements is helpful for the system stakeholders and developers for the better understanding of the system. Due to the generic nature of NL requirements, they can be used for different purposes. Furthermore, requirements of types such as functional and non-functional requirements are also expressed in NL form.

2.4.2. NL in System Specification

NL is usually not reckoned for representing the system specification because of the potential problems such as ambiguity, inconsistency and incompleteness. However, it is also understood that some of the requirements are almost impossible to be represented in formal specification languages, to illustrate this let us consider the following example, "The door state of the system can be easily maintained" [22]. Considering this example due to the ambiguous and incomplete nature of the statement, the developer is unsure about the formal representation of the requirement. Therefore, requirement templates are considered as an option in order to handle the problem associated with NL requirements.

2.4.3. Testing Against NL Requirements

Writing test cases without processing NL requirements may result into non deterministic results. Hence, effect the functional safety of the overall system. Therefore it is necessary to use Natural Language Processing (NLP) techniques in order to address the issues of NL requirements. It is also important to note that requirements driven from requirement template are easily processed by NLP techniques. After applying NLP techniques to the informal user requirements, the task is to come up with test scenarios based on system specifications.

2.5. Correction Cost of Requirement Defects

Problems with requirements specifications can result into the rework. Fixing the problems in requirement specifications can result into high cost. According to an estimation, resolving the requirement errors approximately cost between 30 to 50 % of the actual development cost [23]. The Figure 2.1 depicts that correcting the failures latter is very expensive than to resolve at the earlier stage [24]. Avoiding and pointing the errors early thus have a greater advantage as they minimize the rework.

Test case generation in the software development life cycle (SDLC) is a difficult phase. It takes almost 40 to 70 percent of the overall development cost. If not taken critically then the cost of fixing goes above [18].

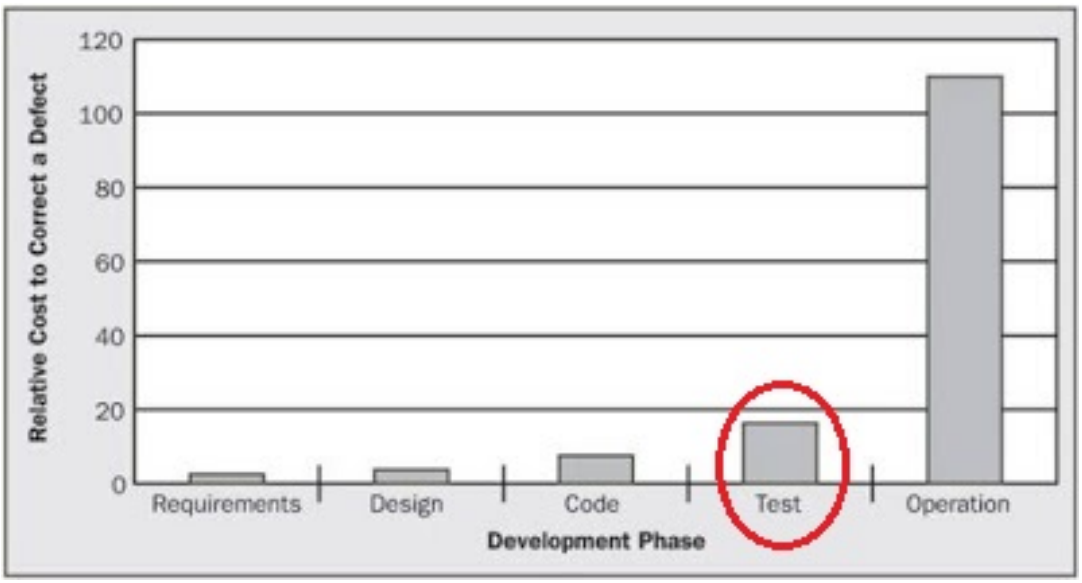

Figure 2.1.: Relative cost to correct a requirement [24]

2.6. NL Requirement Problems

2.6.1. Ambiguity

Ambiguous requirement specification is a great problem for the system developers and testers. One of the sign of ambiguous requirement statement is that a reader could easily misinterpret a statement. Other symptom is that different readers reading the requirement may result into different understanding of the system. The Table 2.1 given above shows different terms which causes the ambiguity. Furthermore, some bad examples of requirements containing ambiguous terms can be found in appendix.

If the requirements specifications are ambiguous then the developers waste their effort in implementing a solution for the wrong problem. On the other side, test cases based on the ambiguous requirements are not deterministic. Representing these requirements in a controlled NL and constructing prototypes are good methods to find and resolve ambiguities [25].

Table 2.1.: Ambiguous terms and how to improve them

Ambiguous Terms	**How to Improve Them**
acceptable, adequate	Determine what comprises acceptability and how it can be judge by the system.
at least, at a minimum	Describe the minimum and maximum tolerable value.
between	The end points must be included in a range
depends on	The nature of dependency must be described very clearly. Check if the system is getting inputs from another system before processing.
fast, rapid	The minimum acceptable speed should be specified for the system to operate.
maximum, minimum	Define the minimum and maximum allowable values of some parameters
flexible	Explain how the system alternates its behaviour based on some changing states.
efficient	State how systematically system consumes resources, how swiftly the system performs its action
several	Describe how many, or state the minimum and maximum limits.
shouldn't	Specify the requirements in positive manner and define what the system will do.
sufficient	Define how much of a thing comprises sufficiency
simple, easy	State features of the system that fulfills the customer's usage expectations.
robust	Describe how system tackle the unwanted or exceptional conditions.

2.6.2. Incompleteness

The requirements specifications are said to be incomplete if an information is missing. It is a hard task to find out which information is not there. Paying attention to the user activity rather than on functional requirements help in avoiding incompleteness. Before processing the requirements (development or testing), these gaps needs to be resolved. Example of an incomplete requirement can be found in appendix Example 1.

2.6.3. Inconsistency

If the requirements of similar type, system, or user requirement collide with the other requirements, this problem is known as inconsistency. Due to the non deterministic aspect of inconsistent requirement, one cannot find which requirement is valid. Requirements which contradict with its parent requirement are also not correct.

2.6.4. Unverifiable

If a requirement is ambiguous, inconsistent and incomplete then it is also not verifiable. Test cases or various verification methods are devised to check if the requirements are properly implemented or not.

The NL requirements are not verifiable if written in long paragraphs. Therefore the requirements are represented individually, properly maintained and labeled with identifiers for the complete traceability.

2.6.5. Redundancy

Duplicated requirements are not rare in requirement specification process, in such cases a particular requirement or a part of requirement is repeated at multiple places in a document. Although the requirements can be placed on multiple locations in a requirement document, however duplication can cause maintenance problems. It could be the case that a modifier updates the requirement document but leaves the other redundant requirements unchanged, hence causes the inconsistency problem in the requirement. This inconsistency needs to be addressed once the defects in the requirements are found [26]. Following it is explained how to deal with the problem of redundancy.

- **Cross Referencing:** This approach allows the cross reference of all text at different locations. It provides the unrestricted use of different size of information.
- **Hyperlinks:** The hyperlinks are placed in the same document or the other document, referring to the origin of the information where it is located.

- **Traceability Links:** Requirements information is saved in the database, these requirements are managed by tools such as IBM Rational DOORS. The requirements saved in the database have a unique identifier. Using the traceability link one can create logical link between the requirements of same type or the other.

3. State of the Art

3.1. Requirement-Based Software Testing

The software development process has phases such as requirement, design, testing and maintenance. Products delivered during each phase are hardly traced with each other. This is not a traditional method of validating the system as there is no traceability mentioned between the requirements and tests.

Requirement based testing involves the verification and validation of the requirements, it also includes the test matrix definition in which requirements are compared with the test cases and number of test cases are obtained at the end. In this activity, review and inspection approaches are of great significance in order to verify requirements and to recognize testable requirements [10].

In Figure 3.1, the process of testing is illustrated with respect to the V-model of software development process. Considering the waterfall model in which software development process is demonstrated in a sequential flow. It is interesting to see that the classical waterfall model reviews testing as the end task while the V-model contemplate testing from the beginning. During the requirement specification stage, based on the black box techniques the test process identifies the test cases and test objectives are also recognized.

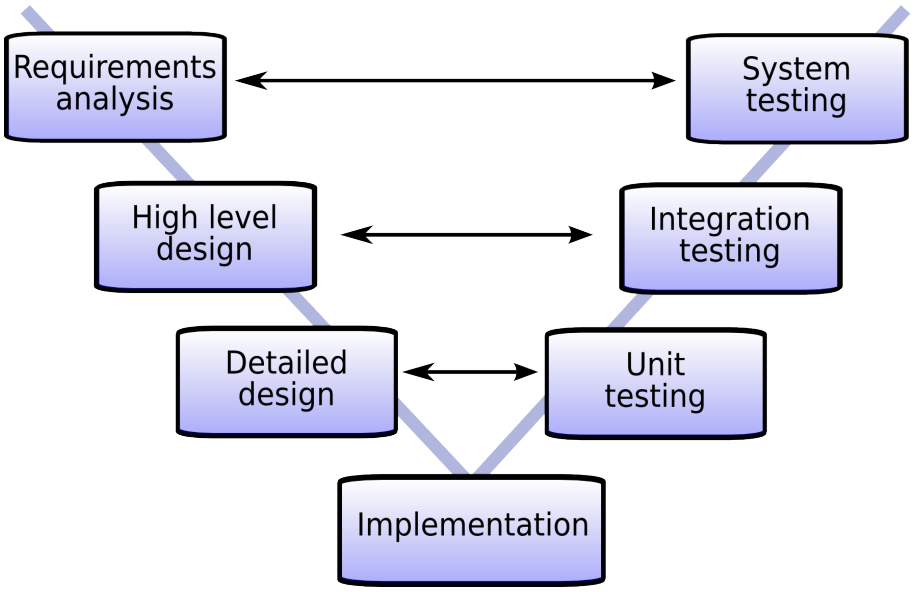

Figure 3.1.: The V-model [10]

3.1.1. Requirement-Based Black-box Test Generation

Usually the system specifications are composed of individual requirements that are represented in informal textual template. These requirements may be ambiguous and difficult to understand. Thus, the individual requirements are accompanied with a formal illustration with the help of language such as specification description language (SDL) [27]. Representing the textual requirements with formal description greatly assist the system engineers to minimize issues of ambiguity, misinterpretation and misunderstanding.

In the approach proposed by [18] individual requirements are illustrated in SDL. The automated system models are created from the individual requirements. The individual requirements are depicted into system models that in the end generated test case based on the individual requirements. This approach had remarkable advantages in terms of minimizing the test cases generation while protecting high quality of requirement driven test suites.

SDL Overview

The specification description language (SDL) is a formal modelling approach regulated by International Telecommunication Union. The purpose of SDL is to eradicated the ambiguities and misunderstanding associated with the requirements. The SDL is used for the specification of systems such as embedded, medical applications and satellite communication [18]. The Figure 3.2 shows the graphical representation of subset of SDL symbols.

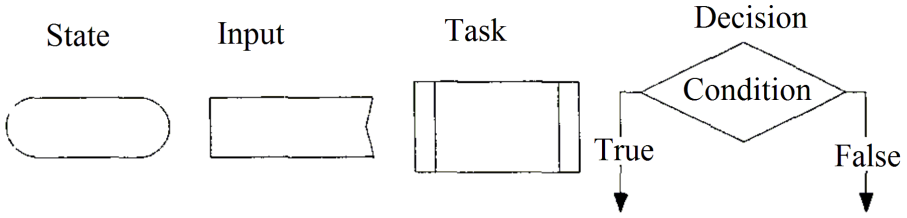

Figure 3.2.: SDL Symbols [18]

System Requirements Representation using SDL

Consider an example in which each individual requirement explains the functional behavior of the system. These requirements are denoted with ID , their textual explanation and their comparable SDL illustration. However, the two kind of descriptions are equal. The dual description helps the testers, developers and managers for better understanding of the system. An example shown in the Figure 3.3 portrays the dual type of individual requirement representation. Whereas, the class denotes whether the requirement is security critical or not.

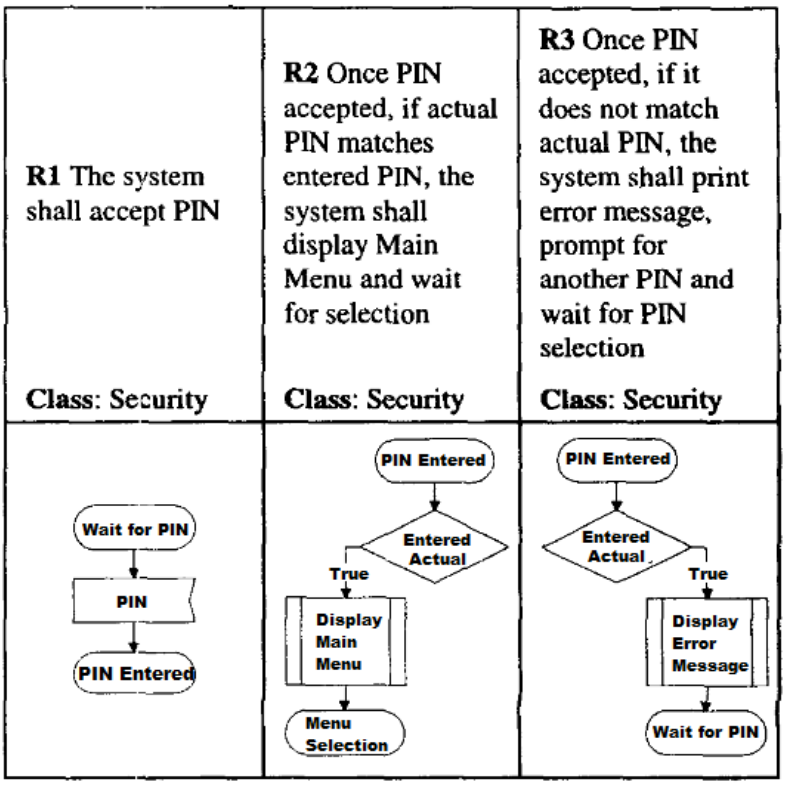

Figure 3.3.: Samples of Individual Requirements [18]

In this section the individual requirements and its formal description are used to form the SDL system model. The system model is then converted into extended finite state machine (EFSM). This approach is automated but it is restricted as controlled requirement is used for the formalization of system specifications. Thus, for more complex and dynamic requirements, this method will not be helpful.

3.2. Use Case Driven Approach for Testing

Model based software testing techniques are already used for the requirement validation [28], [29]. In the approach [17], the concentration was towards universally accepted exercises based on universal modelling language (UML) in order to aid the object oriented development process. This approach proposes to generate the test scenarios automatically from use cases in the context of object oriented embedded software. The problem of traceability between the use cases and concrete test cases were also considered.

In the use case driven approach many ambiguities associated natural language requirements are untangled. The UML uses cases were strengthened with the contracts. The contracts are pre and post conditions attached with use cases [30]. In this manner, when requirements are expressed using use cases and contracts, they

can be simulated orderly to find their correctness and consistency. The aim of this method is to reduce the burden by automating the task of test generation and shift the effort towards the requirement specification task.

The figure 3.4 depicts the two-stage approach to automatically produce functional test cases from requirements specifications. In the first stage from step (a) to (c), the goal is to produce test objectives from use case aspect of the system and in the second phase from step (c) to (e), the goal is to produce test cases from the test objectives.

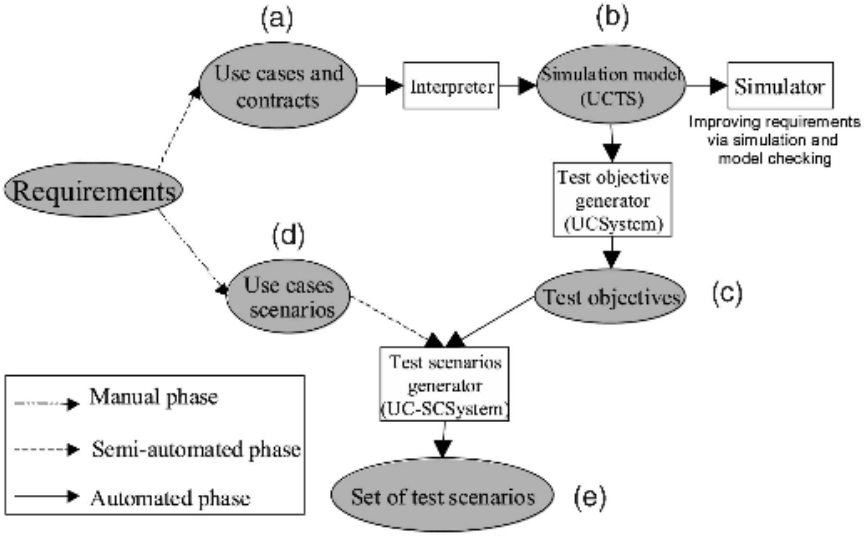

Figure 3.4.: Global methodology for requirement-based testing [7]

3.3. Generating UML Models from Natural Language Requirements

Since, there exist approaches which generate the test cases automatically from the formal methods such as sequence diagram, statecharts, collaboration diagram etc.. As the formal models are created manually therefore can be error prone and are expensive to resolve later in the software development stage. In the approach proposed by [31], a semi-formal method was proposed to assist the developers in order to produce UML models from normalized NL specifications with the help of natural language processing (NLP) techniques.

As the requirements expressed in NL are ambiguous, inconsistent and incomplete, it is usually the responsibility of requirement analyst to detect and fix these potential problems. A developer having less domain knowledge may overlook such defects in user requirements, hence resulting into non-deterministic interpretation of the

system. There exist approaches to reduce the gap between requirement analysis and design stage in terms of developing object oriented models from NL requirements.

3.3.1. The UMGAR Approach

The technique named as UML Model Generator from Analysis of Requirements (UMGAR) is used to support the designers and requirement analyst in generating UML models from NL requirements using advanced NLP methods. The proposed technique can generate use-case diagrams, design class model and collaboration diagram [31],[32]. Figure 3.5 shows the process architecture of UMGAR which consists of two parts. The first part consists of normalizing the requirement components (NLP Tool layer) and the second part is based on model generator component.

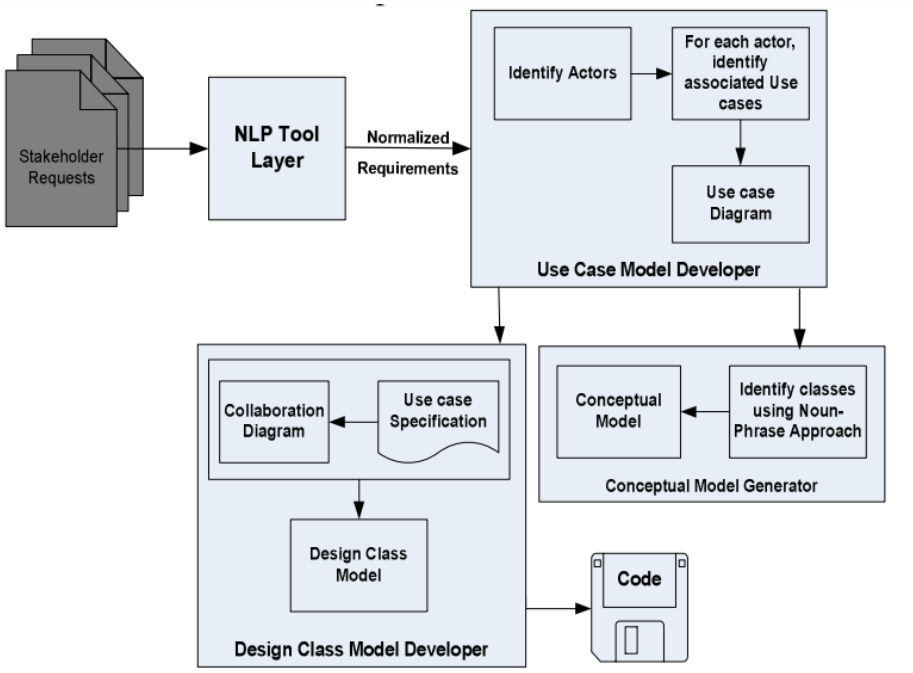

Figure 3.5.: The process architecture of UMGAR [31]

3.4. Minimizing Ambiguity in NL Requirements Specifications

Traditionally, natural language (NL) is being used for the requirement specification. According to an estimation its usage is about 71.80% in documenting requirements [1]. Still the requirements are generally ambiguous. For the automated software requirements modeling or test generation it is necessary that the requirements remain unambiguous. A technique named as Semantic of Business Vocabulary and Rule

(SBVR) was presented to obtain controlled representation of requirements specifications. The task of SBVR is to produce software requirements more accurate and consistent. This technique makes the requirements machine understandable, since it is based on higher order logic [33]. Following are the features of SBVR which help in the generation of controlled representation of English [34]:

1. Controlled formalization based on rules
2. Semantic formalization of Natural Language
3. SBVR Formal Notation

3.4.1. Translating NL to SBVR

Here it will be described, how the natural language text in English mapped to SBVR depiction as well as obtaining object oriented information from SBVR description. This is illustrated in the Figure 3.6.

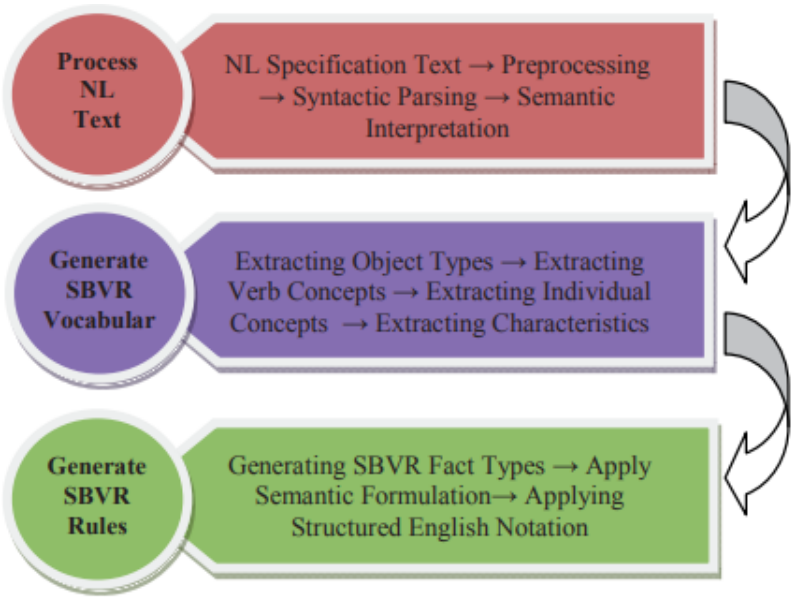

Figure 3.6.: Translating Framework for NL to SBVR [34]

3.4.2. Documenting Requirement Using NL Boilerplates

The approach presented in [21] focuses on the Requirement Engineering (RE) stage in the life cycle of software development. This phase gives the opportunity to the system developers to sufficiently understand the requirements of system stakeholders [35]. Moreover, it is a vital activity, considering the context of generation of test cases based on requirements specifications in NL.

Although there are potential problems associated with NL requirements such as inconsistency, ambiguity and incompleteness [12]. In order to avoid these problems from NL requirements. An approach was proposed in which requirements specifications from the stakeholders should be documented in a restricted manner [16]. This approach was considered helpful, particularly for the beginners in order to represent the requirements in a standardize way. The boilerplates represent two kinds of requirements such as functional and non functional. Following Figure 3.7 depicts the process of requirements elicitation with the help of natural language boilerplates.

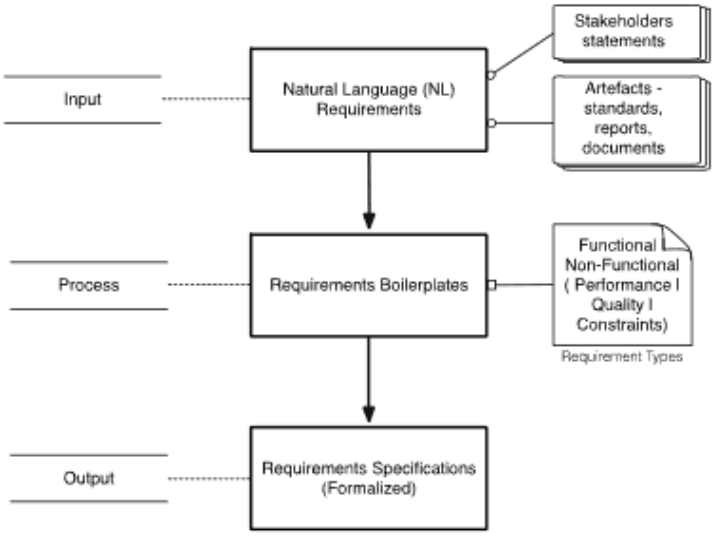

Figure 3.7.: Expressing requirement specifications using NL boilerplates [16]

The following are different example of boilerplate templates for system interface and autonomous requirements [16].

1. Interface
   - The "system name" shall/ should/ will able to "process"
   - The "system name" shall/ should/ will able to "process" "object" "object information"
2. Autonomous or independent
   - The "system name" shall/ should/ will "process"
   - The "system name" shall/ should/ will "process" "object" "object information"

3.5. Machine Learning Technique for Requirement Specification

Previous approaches such as structured natural language and normalized natural language have already been used to remove ambiguity, inconsistency from natural language requirements. However, the issue of incomplete requirements specifications still remains [36]. Considering the existing approaches in which natural language processing techniques are used with search based software engineering methods, the idea is to replace human based search techniques with machine based search approaches [37].

In order to understand this technique, consider two end users: trained user and untrained user. The untrained user has no specific domain knowledge of system modeling languages and the trained user has relevant experience in formal modeling languages such as sequence diagram or UML use cases. Using the machine learning approach, it is explained how the effective formal system specifications are generated from incomplete and inconsistent requirements [38]. The NL specifications from untrained users are not formal, therefore extra resources are required. The workflow of untrained user is shown in Figure 3.8.

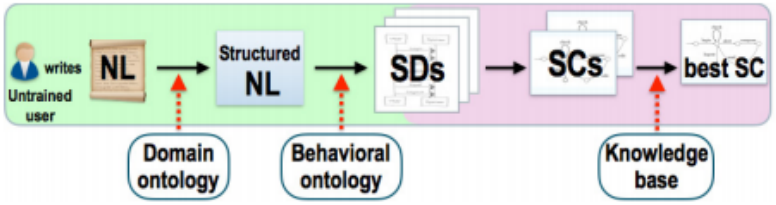

Figure 3.8.: Untrained user worklflow [38]

The machine learning approach for requirement specification demonstrates two workflows for untrained and trained users, these workflows are running in parallel with each other. These workflows are the building blocks of the overall system. A greater number of information is gathered over many runs of the system and the requirements specifications are refined after each run. The information collected from trained user workflow assists the untrained user workflow [38]. The Figure 3.9 shows the cross links between the two workflows, whereas black arrows represent different phases of workflows, red arrows illustrate additional resources being used after every stage and the dotted blue arrows show different levels.

The future of machine learning approach is to obtain behavior ontology and convert structural natural language into sequence diagram. However, the focus is towards automating the entire process and domain information is needed for improvements.

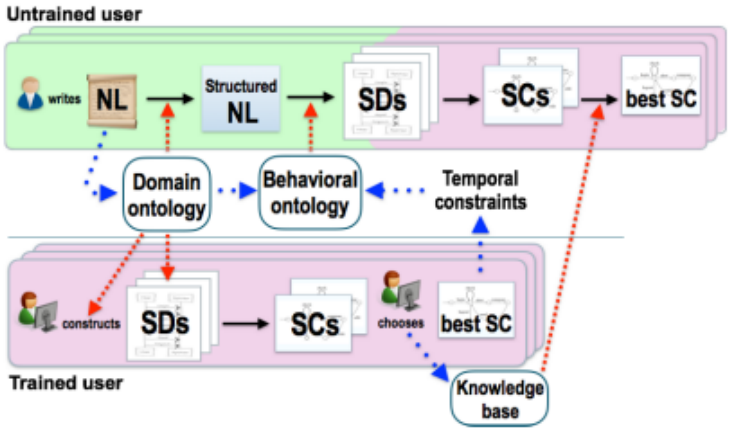

Figure 3.9.: Cooperation between two workflows [38]

3.6. Relevant Work at Professorship Computer Engineering (TU Chemnitz

Relevant work has been done by professorship Computer Engineering (Chemnitz University of Technology) such as automatic generation of tests especially for education purpose in the field of software engineering. In this research a concept was presented in order to generate questions and tasks automatically in the domain of software engineering. The approach was established on a knowledge base, accommodating architectural knowledge such as functions, modules etc [39].

An approach was presented in [40] to test AUTOSAR software efficiently build on automatically generated knowledge base. In this approach, high quality AUTOSAR development was made certain by automatically examining the configuration files and source code of a particular project before compiling. The proposed approach was independent of any tool chain employed and the version of AUTOSAR. In the approach proposed by [41], in which the AUTOSAR basic software was evaluated on present-day ECUs using an application. Since, it is difficult to cover tests for all feasible combinations of configuration parameters from various AUTOSAR modules without taking into account particular application features. The proposed technique automatically generated test cases from AUTOSAR configuration files with the goal to assist integrating AUTOSAR software on a particular ECU.

3.7. Analyzing Existing Approaches

3.7.1. Criticism

As shown in this chapter, test cases can be generated automatically using the formalization techniques such as UML statecharts, sequence diagrams, finite state machines etc. However, formalization of system specifications is time consuming, expensive

and require domain knowledge [19]. The formal interpretation of the system is sometimes non-deterministic, as each expert converts the system specifications based on individual understanding. Therefore, testing against the non-deterministic formal representation of system specifications could effect the deterministic and traceability aspect of the test cases.

3.7.2. Way Forward

This thesis will address the generation of test cases based on NL requirements. Testing of the system or planning test case generation at the early requirement stage is advantageous as it assists the system designers and analysts to get a good overview of the system and communicate the requirement. It ensures the requirement testability, as the verification and validation cost of the system is greatly reduced. While going through the requirement documents, a system tester should recognize each action and state in a requirement. The test cases are extracted from these statements. A statement expressing the functional or non function specification of the system is a prospective test case [10].

The NL requirements are semantically inconsistent and syntactically unclear. Hence it is of vital importance to handle such issues before testing. Therefore, this thesis will also specify the current state and challenges related to testing against the natural language requirements. In terms of proposed solution, controlled NL requirements are extracted with the help of proposed requirement templates. Furthermore, these requirements are processed using NLP techniques in order to understand the requirements. The Figure 3.10 shows the conceptual view of the generation of automated testing from NL requirements.

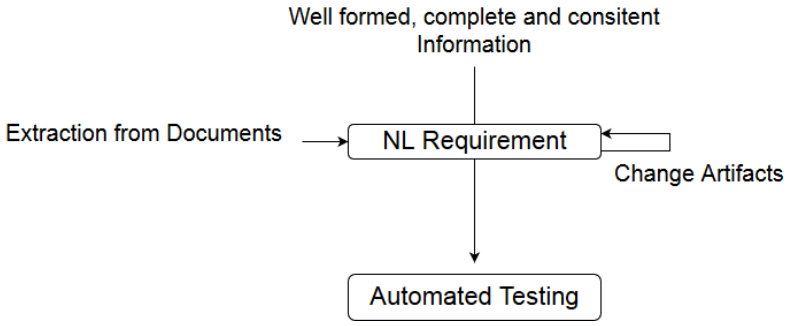

Figure 3.10.: Overview of NL requirement based testing

3.8. Summary

The Chapter Basics provided an overview of problems related to manual software testing and described why NL requirements are considered in this research. It was

also explained that transformation of NL requirements into formal modelling specifications (UML, sysML etc.) can help in the automatic software testing but these specifications may not cover non-functional specifications of the system. The role of NL requirements in requirement elicitation, system specification and requirement based testing was also discussed in detail. Finally, issues associated with NL requirements such as ambiguity, incompleteness, inconsistency, and redundancy were explained in detail.

The Chapter State of the Art shown approaches to generate test case from formal representation of system specifications. It was found that object oriented modeling techniques are frequently inaccurate [18]. In the perspective of this thesis, the task is to represent the requirements in a formal representation so that they could be easily processed by the machine. This can lead towards the easy automation of test cases. Techniques such as requirement based testing and use case driven approaches for testing were presented in this chapter. The requirements in these approaches require additional description such as SDL and contacts (pre- and post conditions). However, these methods were semi-automated and semantics provided to the use cases are precise and rigorous. This chapter also covered the aspect of generation of UML model from NL requirements. Such approaches are not fully automated and the limitations of NL requirements are always present [17]. The details of control NL requirements were also studied which proved helpful in term of defining the templates for the requirements. Finally, a machine learning approach was presented for the formal representation of system specifications using trained data from domain experts[38].

4. Concept

This chapter will explain the proposed solution to the problem statement. Here, we will discuss the systematic approach in order to generate test cases from NL requirements, as well as describe how the natural language processing (NLP) techniques are used in order to process the informal textual requirements to the formal representation of system specifications.

The development process model called V model is followed during the systematic approach. In this thesis we have started from the top left hand side of the V model from requirement analysis to high level system design. The task is to generate the integration tests from this high level design based on requirement specification. The Figure 5.1 shows the proposed solution in terms of V model of development.

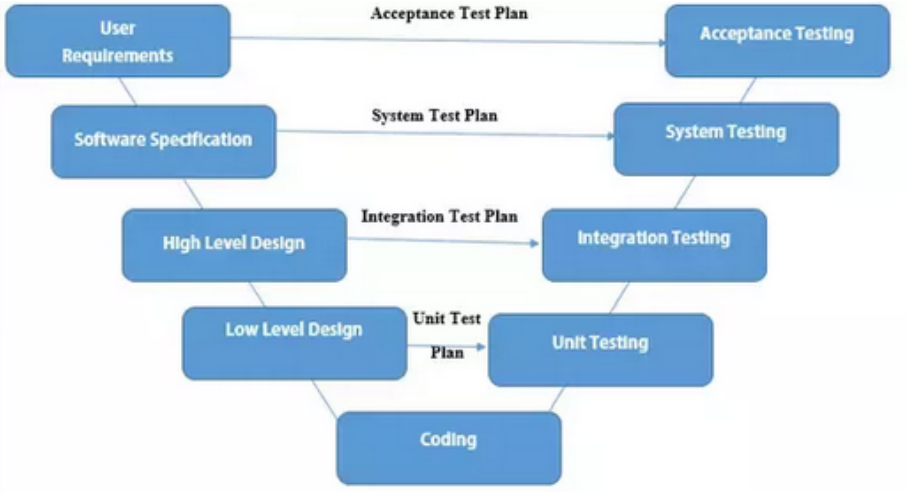

Figure 4.1.: V Model in perspective of proposed solution

4.1. Systematic Approach Overview

A systematic approach is followed during the accomplishment of suggested solution, we have started from the top left side of the V model then moved downwards to the conceptual details about the system requirements and at the end generating test case based on system requirements. The steps pursued during implementation of solution are as follows.

1. Gathering NL requirements

2. Conceptual system design
3. Specifying signals for proposed system
4. Extracting requirement phrase structure
5. Processing requirements statements
6. Generating test cases from template-based requirements

4.1.1. Gathering NL Requirements

In the first step, the system specifications are collected in informal textual requirements. Since the NL requirements have issues such as ambiguity, inconsistency and incompleteness. It is important to express these requirements in the form of control structure. Following table shows the problems in NL requirements specifications and their proposed solutions.

Table 4.1.: NL Problems and Solution Proposed

Problems	**Solution Proposed**
Incomplete Requirement	Record the important information such as pre conditions, input values, constraints, output values and post conditions. The missing information shall be indicated with TBD standard flag.
Inconsistency	In order to make the requirements consistent it is necessary to label the requirement. There are three different labeling techniques such as sequence numbering, hierarchical numbering and textual tagging. A logical representation of requirements prevent conflicts e.g Parent-Child requirements prevent conflict between the requirements.
Ambiguity	Avoid complicated logic instead divide the requirement into multiple consistent statements. Always use positive sense while documenting the requirements and prevent negation from functional requirements. Do not use the synonyms, near synonyms, abbreviations, and ambiguous wordings such as acceptable, efficient, several etc.

4.1.2. Conceptual System Design

To clarify the system requirements, normally the system stakeholders illustrate a conceptual design of the proposed system. The importance of presenting this design is to explicitly portray the following aspects of the system:

- System actors

- Basic flow of the system
- Input processing and output generation

Following example in the Figure 4.2 depicts the abstract representation of a system that takes input from another system as well as from the other sources. Furthermore, an output is generated after processing the input values.

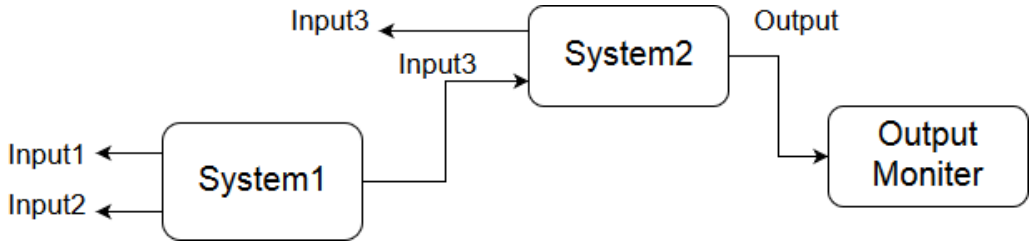

Figure 4.2.: Abstract system design based on user requirements

4.1.3. Specifying Signals for Proposed System

Since number of messages are transported from one part of the system to the other. Therefore, it is necessary to define the unique name for each message. The message signal data type plays a vital role in defining the range interval, therefore it is convenient for the system analyst to denote the ranges (maximum and minimum values) stated in a requirement statement. Furthermore, it is necessary to mention the source of input signal and the destination of the output signal. A short description about each individual signal is a good practice to explicitly describe the functionality of the system.

In the following Table 4.2, detailed information about the attributes of signal is provided. The signals are expressed in a document called signal lexicon.

Table 4.2.: Proposed system signal attributes and their description

Signal Attributes	**Description**
Signal Name	Provide a unique name to each signal. If the signal name consists of two words then it is a good practice to place an underscore between them. The signal name could be defined using alphanumeric characters
Data Type	A signal data type is an attribute which indicates the system how it will be interpreted. Furthermore, it determines the operations perform on the data, the purpose of the data, and the constraints on the data. Some commonly used data types in a signal lexicon document are Unsigned char, Signed char, Boolean etc.

Signal Attributes	Description
Algorithm	In this column of the signal lexicon document, specific signals are set against a particular value, for example initial value is represented as "init" which is set to zero (init=0).
Replacement Value	A substitute signal value is defined which is sent to the system in the absence of a defined signal value.
Remarks	Elaborate about the purpose of each signal.
Initial Behavior	Defines what value is sent to the system when it is initiated
Sender	Source of the signal
Receiver	Destination of the signal

4.1.4. Extracting Requirement Phrase Structure

NL requirements are carefully examined in order to make them complete, accurate, unambiguous and verifiable. Requirements statements are processed and then converted into phrasal structure. The following points represent the key characteristics in order to extract phrase structure from NL requirement.

- To satisfy the standard of refinement and traceability, individual requirements are assigned with a unique and continuous identifier.
- A controlled structure of a sentence is defined which helps in managing the NL requirement ambiguity.
- The technical terms stated in the requirements should be listed in the project's glossary.
- Take out only the direct relations but avoid indirect ones.
- To avoid major requirement problems, requirement style elements are defined which state the techniques to manage ambiguities.

4.1.5. Processing Requirement Statements

Generally, extracting or gathering requirements information from the system stakeholder, reports, standards and other associated documents are all represented in NL requirement statements. The elicited requirements are further segmented and processed based on the requirement template explained in the figure 4.1. Throughout this process, the informal requirements are documented by following a specific template with controlled vocabulary. As a consequence, the requirements specifications are formalized in a structured fashion. The following Figure 4.3 depicts the conventional process of transforming NL requirements specifications into formal requirements specifications using requirement templates.

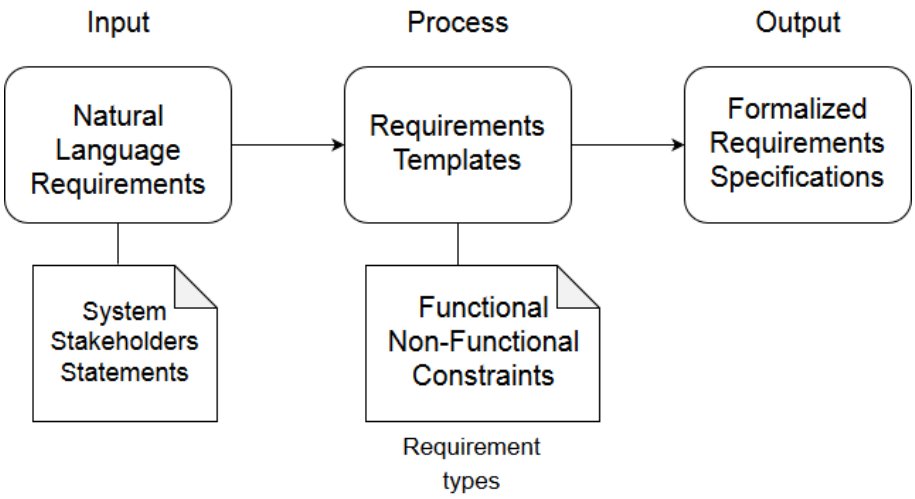

Figure 4.3.: Extracting formal requirements using requirement templates

Requirements Pre-processing

After extracting the phrasal structure through NL requirements. The task is to normalize the text in which the unnecessary words are removed and stemming is performed [42]. However this process is based on domain knowledge and the capability of parsing algorithm to acknowledge the sentence tags in a certain format [20]. In order to illustrate requirement Pre-processing, consider the following example of a NL requirement statement.

"The system shall take unsigned char values from an interval [0, 255]"

Following steps are taken in order to normalize the given requirement statement.

- Join all the adjective with their nouns such as "unsigned char values" by placing a hyphen between them to "unsigned-char-values".
- Usually, range interval is specified as for example"[0,255]" but for better processing of a statement by NLP methods, place a white space between square brackets and numerical values such as "[0 , 255]".
- Articles such as "The", "A" is "An" should be removed from requirement statement. If they show up in some formula, then an appropriate replacement should be added.

The requirement statement after pre-processing is given as follows:
"System shall take unsigned-char-value from interval [0 , 255]."

Parts of Speech (POS) Tagging

For each word in a requirement statement a POS tag is assigned using NLP POS tagger. It is a convenient method for information extraction, understanding of word

sense as well as helps in word parsing by associating a label to each word [43]. Figure 4.4 portrays the example of POS tagging of normalized requirement statement.

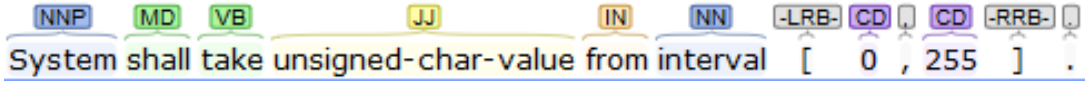

Figure 4.4.: POS tagged requirement statement

After the identification of POS tags in the above given requirement statement, it is appropriate to describe each tag in the following Table 4.3.

Table 4.3.: POS tags description

Tag	**Description**
NNP	Proper Noun
MD	Modal verb
VB	Base verb
JJ	Adjective
IN	Subordinating conjunction / Preposition
NN	Common Noun
LRB	Open parenthesis
CD	Cardinal
RRB	Close parenthesis

Sentence Parsing

A requirement statement is provided as input to NL parser, which identifies the syntactic tree structure that helps in the understanding of the sentence [44]. Figure 4.5 shows the parsing tree structure of requirement statement depicted in the Figure 4.4.

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD shall)
      (VP (VB take)
        (NP (NN unsigned-char-value))
        (PP (IN from)
          (NP
            (ADJP (JJ interval)
              (NP (NNP -LSB-) (CD 0) (, ,) (CD 255)))
            (NNS -RSB-)))))
    (. .)))
```

Figure 4.5.: Sentence Parsing

Phrase Level Dependencies

After retrieving useful information from each words in an individual NL requirement statement using POS tagging, the statements are interpreted using NLP parsing methods. In this section, general dependencies are raised to semantic units, which is helpful in determining relationship between entities in a requirement statement [45]. The basic overview of the systematic approach from NL requirements to phrase level dependencies is illustrated in the Figure 4.6.

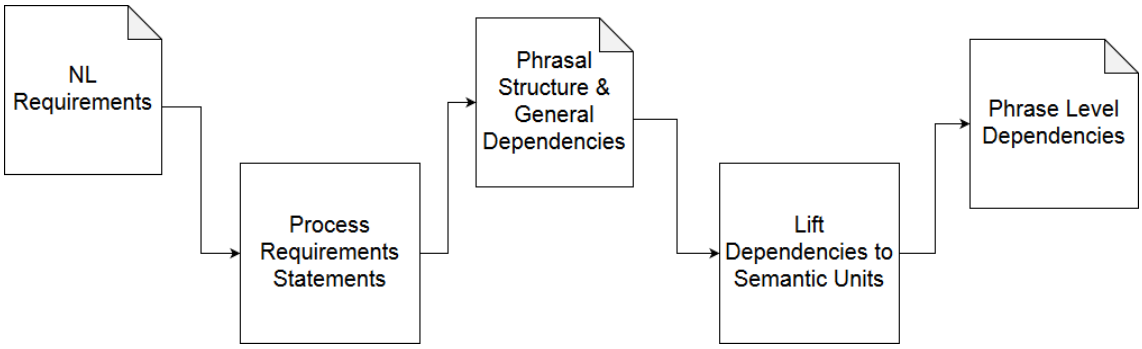

Figure 4.6.: Systematic approach to extract phrase level dependencies from NL requirements

Different core NLP approaches are used to identify basic dependencies in a sentence as well as to extract binary relations from a plain text. However, for the recognition of words in a sentence it is necessary to keep the sentence grammatically correct. Figure 4.7 shows the dependencies of each word or group of words on each other. For this purpose we have used dependency parsing techniques such as Basic Dependency and Open Information Extraction (Open IE).

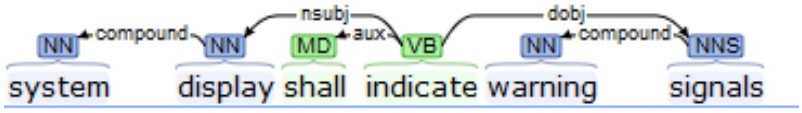

Figure 4.7.: Basic dependency example

To demonstrate the relationships of entities in a requirement statement, dependency parsing approach such as Open Information Extraction (Open IE) is used. However, Open IE approach is not capable of correctly recognizing relations in a complex requirement statement. This is portrayed in the Figure 4.8.

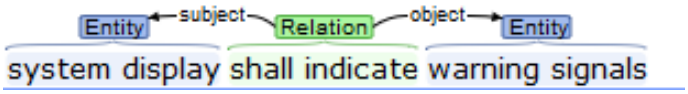

Figure 4.8.: Open Information Extraction example

Constructing Entity Relational (ER) Diagram

After recognizing the entities such as subject, object, and the relation between them, a system can generate a model from a processed requirement statement.

Figure 4.9 portrays the two entities represented as rectangles and the relation between the two entities is mentioned in the middle while the arrow specifies the flow from one entity to another

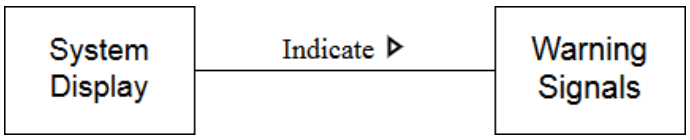

Figure 4.9.: Modeling relationship between entities

4.1.6. Generating Test Cases from Template-based Requirements

Till now, this thesis describes the techniques such as processing the template based requirements using NLP approaches and recognizing the dependencies between each word or group of words in a statement.

The thesis focuses on the generation of test cases based on requirement templates. Therefore in terms of producing test cases from requirement templates, one need to identify as well as specify the constraints from individual requirements. At the end, this thesis will propose a boundary value analysis approach for test case generation.

Constraints Identification

After the identification of POS tags for each word in a requirement statement our task is to check whether the conditional statements are also identified by the dependency parsing techniques. Furthermore it is also necessary to check if constraints defined in a statement are specified clearly or not. Let us take a following example in which a requirement statement is represented in a controlled structure using a requirement template.

"System shall display warning-message if speed is greater than 150 km/h"

From the given example, the identified constraints do not answer the following questions:

1. What the system will display when the speed is less than 150 km/h?
2. What the system will display when the speed is exact 150 km/h?

Considering this ambiguity, it is necessary to specify the constraints comprehensively. Following table shows different examples of ambiguous constraints and how to represent them in a requirement template.

Table 4.4.: Defining and Identifying constraints

Ambiguous Constraints	**Good Practice**
Temperature is low	temperature-range interval [min, max]
System-status is valid	System-status != null
Errors are absent	Error_size == 0

Test Case Generation

After stating the test constraints for the system, test scenarios can be generated based on test constraints. For a large range of data it is practically labour intensive to write test case manually for each scenario. Therefore we require an uncomplicated an efficient approach to efficiently choose test cases from a large pool of ranges in a way that all test cases are comprised. To attain this goal, method such as equivalence class partitioning is used [46].

We have taken following example to illustrate basic dependency using statement parsing, afterwards a technique such as equivalence class partitioning is applied for test case generation.

"System shall take search-characters in range interval [5 , 20]"

Figure 4.10 depicts the basic dependencies of the statement, which identifies the subject, action, object and the scenario in which the action will be valid.

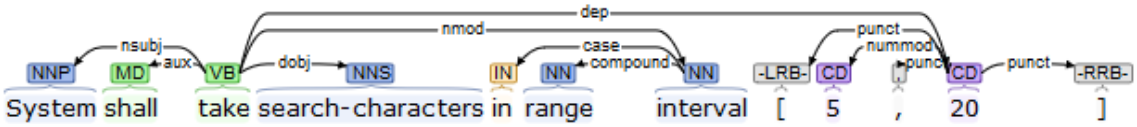

Figure 4.10.: Requirement statement basic dependencies

Consider that it is impractical to state test cases for the whole range as it is time consuming and performed manually. To handle this problem we have applied equivalence class partitioning by splitting the range into chunks which state the behaviour of the system. The test conditions for the example shown in figure 5.10 are stated below. Furthermore, these test conditions are depicted graphically in Figure 4.11.

- Any value greater than 20 is considered invalid
- Any value less than 5 is considered invalid
- Only Values from 5 to 20 are considered valid

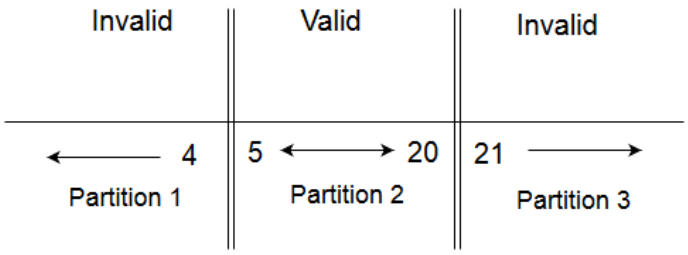

Figure 4.11.: Partitioning ranges into valid and invalid

4.2. Summary

In this chapter a systematic approach was defined to generate formal representation of system specifications from natural language requirements. The ultimate goal of the proposed approach was to generate test cases by means of using template driven requirements. For the realization of the proposed solution, the V model development process model was followed.

The systematic approach defined in this chapter has five 5 steps. In the first step, informal NL requirements were elicited from the system stakeholders. The second step defines the conceptual design of the system driven from user requirements. In the third step, signal were stated for the proposed system. These signals have attributes such as data type, algorithmic values, sender, receiver etc. In the fourth step, semi-formal requirements were processed using NLP techniques. The output of this step is controlled NL requirements. In the final step, a simple and efficient technique was used to smartly select test cases from a large pool of ranges in a manner that all test cases are comprised. To achieve this goal, equivalence class partitioning was used.

5. Formal Requirement Criteria

As the approach proposed by this thesis and presented in Chapter Concept requires a formal written specification, this Chapter provides the criteria of documenting requirements in formal manner based on a specific template.

5.1. Instructions for Writing Formal Requirements

There exist no textbook method for documenting excellent requirements, the more experience you have the better you can specify requirements. For the effective documentation of requirements it is vital to use technical scientific writing instructions and enlist user language.

For the effective documentation of requirements specifications following recommendation were presented [47].

- Specify requirement statements that contain formal spelling, grammar and punctuation. Make sure that these statements remain direct and short.
- The sentences should be expressed in active voice.
- Avoid the use of synonyms and close synonyms. The terms used in a statement should be defined consistently as prescribed in the glossary of requirement document.
- Decompose the high level requirement into adequate detail in order to clarify it and eradicate ambiguity.
- Define the requirement statements in a consistent way, for example "the system shall" then comes the action verb and at the end an observable result is defined. Avoid using the terms "should","might and "may" and similar words as they lack clarity.
- State the specific actors for example "The cockpit view shall...." instead of generic actors.
- The usage of ambiguous terms should be prevented as they might result into unverifiable requirements. There terms are stated in the table 3.1 and this table also describes how we can improve the definition of such vague terms.

5.2. Formal Requirement Process Advantages

It is of vital importance to spent time on collecting, writing, analyzing and managing requirements. Consider the time spent as an investment as it will save a lot of time and cost for the latter activities. Following are the benefits of superlative requirements process [48].

- Reduced development effort.
- Less requirement faults.
- Quick Development.
- Less possibility of misinterpretation.
- Systematic approach to project development.
- System testing based on requirements are more precise.

5.3. Requirement Statement Attributes

There are particular characteristics, which specify the good requirements from those with issues. Following sections will specify different qualities of individual user requirements and system's functional requirements [49],[50].

5.3.1. Complete

The requirements must thoroughly explain the functionality of the system. It should hold every important information for the system developer in order to design and implement the system functionality. If some information is missing one must use TBD (to be determined). Whereas TBD is a conventional flag which notify the missing of information.

5.3.2. Accurate

The individual requirement statement should precisely define the system's functionality. The source of requirement is used to check its accuracy, for example the original user requirements. There should not be a conflict between formal requirements specifications and actual user requirements otherwise requirements will be called inaccurate.

5.3.3. Unambiguous

The requirement statement must have a unique and reconcilable explanation for each reader. The NL requirements are usually unclear, therefore it is advised to express the requirements in straightforward, brief and clear language suited to the user domain. The requirements must be comprehensive which means that the reader could easily understand what is stated in it. Specify the key terms in the glossary in order to prevent the reader from confusion.

5.4. Requirements Labeling

To fulfill the standard of traceability and refinement, each requirement statement must be assigned with a distinct and continual identifier. Labeling the requirements have advantages such as specification of change history, cross referencing or in the traceability environments. In the following sections we will specify the advantages and defects of various requirement labeling techniques [51].

5.4.1. Sequence Numbering

This technique is used for the distinctive sequence numbering for individual requirement such as SRS-12, SR-23 or UR-12. Requirement management tools use such labeling methods for representing the requirements in their databases. The prefix shows the category of requirement, such as SRS represents software requirement specification. If a requirement is removed then the particular number associated with it will not be reused. Following are few limitations of sequence numbering approach.

- Such representation of requirement does not give any hierarchical collection of associated requirements.
- This technique does not provide any information regarding the requirement.

5.4.2. Hierarchical Numbering

It is a traditional approach. The requirements are represented in labels that start with 1.1 (for example 4.1, 5.1 etc). More digits are designated for more description about the requirements and subordinate requirements. Examples of specifying numbering for low-level requirements are 3.2.1, 4.2.1 etc. This technique is concise and compact. These numbers can be generated by the word processor automatically. Following are few limitations of hierarchical numbering approach.

- Representing the requirements in numeric form do not inform about the reason of individual requirement.
- This method does not produce persistent tags.

- Insertion of a new requirement will automatically increase the following requirement numbering in that section
- Deletion or shifting of one or more requirements will automatically decrease or shift the numbering in that section.

5.4.3. Hierarchical Textual Tagging

It is hierarchical labeling based on text for tagging single requirements. Just take an example "The system takes engine value in an interval more than 50 and less than 100". This requirement will be label as System.EngineValue. This tag represents the system and function such as engine value.

The hierarchical textual labels are organized, purposeful and unchanged by addition, removal, or shifting of other requirements. One drawback of hierarchical textual tagging is its large size when compared with the numeric labeling techniques.

5.5. Requirement Template

There are conventional keywords for the recognition of a functional requirements. Functional requirements represent the systems behavior or act under particular conditions or user actions permitted by the system to take.

Natural language requirements specifications extensively use various arbitrary verbs such as would, could, should, will, must, might, shall, may and can. These verbs are causally replaced in a natural language (English). It is up to the reader to recognize the context and derive the understanding of the requirement specification. For example, Does "must" has different implication than can". Typical verbs such as "shall", "must" and "will" separate the requirement from the other data in a particular document.

The requirement templates depicts the specific structure of a sentence which helps in resolving the ambiguity of NL requirements. Figure 5.1 shows the structure of a requirement template and an example to prove the case.

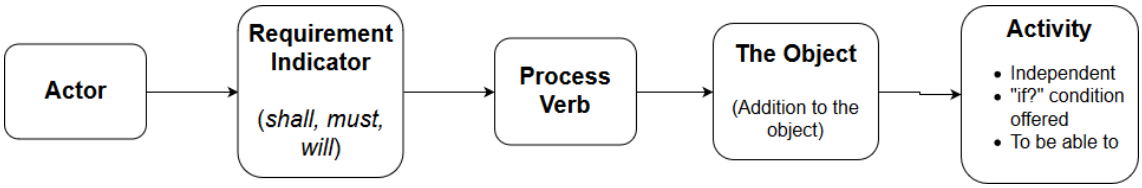

Figure 5.1.: Requirement Template

The above given example shows a requirement statement derived from defined template.

"The System shall indicate warning message if door state is open"

- *"The system"* is the **Actor**
- *"shall"* is the **Requirement Indicator**
- *"indicate"* is the **Process Verb**
- *"warning message"* is **The Object**
- *"if door state is open"* is **Activity (condition)**

5.6. Requirement Style Elements

The major challenge dealing with natural language requirements is ambiguity. There exist no simple formulaic technique for requirement specification. The superlative requirements should have characteristics such as conventional grammar, correct spelling, well organized structure and a reasoned cooperation.

5.6.1. Some Random Rules

Following are some rules in terms of writing requirements. These rules have been examined during the analysis of various software requirement specification documents [52], [53].

- Avoid using the word "and" in a requirement statement. This particular word shows that there are two requirements which needs to be separated.
- There should be a single sentence in a requirement.
- There should not be more than 22 words in a requirement statement.

5.6.2. Perspective Based Requirements

There are different ways in order to express the functional requirements. Some of the requirements are based on system perspective as their focus is to create a system dependent on functional requirements. While on the other hand, user requirements illustrate how the user operates with the system. Following are the universal structures of requirements written from system and user perspective [51],[54].

System Perspective Requirement Structure

- **Conditions:** "if [some constraints are true]"
- **Output:** "... the actor shall [perform something]"
- **Qualifier:** "....[reaction time target or quality aim]"

User Perspective Requirement Structure

- **User Type:** "The [User Classification or name of actor]"
- **Output Type:** "... shall be capable to [perform something]"
- **Object:** "....[to something]"
- **Qualifier:** "....[reaction time target or quality aim]"

5.6.3. Hierarchical Requirements

Hierarchical requirements are represented in parent and child fashion. There is one parent requirement and one or more than one child requirements. The specification of parent requirement is fulfilled when each of the child requirements are satisfied [51].

The parent requirement is usually represented in title or heading and the child requirements are combined to fulfill the parent requirement. This shown in the following example.

1. HMI Navigator
   a) The driver shall input the destination location on the user interface
   b) The system shall validate the input depended on the coordinate constraints.

5.7. Managing Ambiguities

As discussed in Chapter 2 about the problems caused by ambiguous requirements and terms which cause them. Here we will discuss about the various source of ambiguities and what improvements could be made to remove such ambiguities. Furthermore, appropriate examples are provided for better understanding.

5.7.1. Complicated Logic

A complicated Boolean logic can easily result into ambiguous and incomplete requirements [54]. Let us take the following example into examination.

"If the value of fuel tank sensor is less than or equal to 10 liters then set the HMI indicator light to blue, otherwise if the fuel tank sensor value is greater than or equal to 40 liters then set the HMI indicator light to red."

This example has few conditions covering a certain range, however it did not cover the comprehensive range such as, fuel level value greater than 10 or less than 40. This scenario is depicted in complex logic decision tree in the following Figure 5.2.

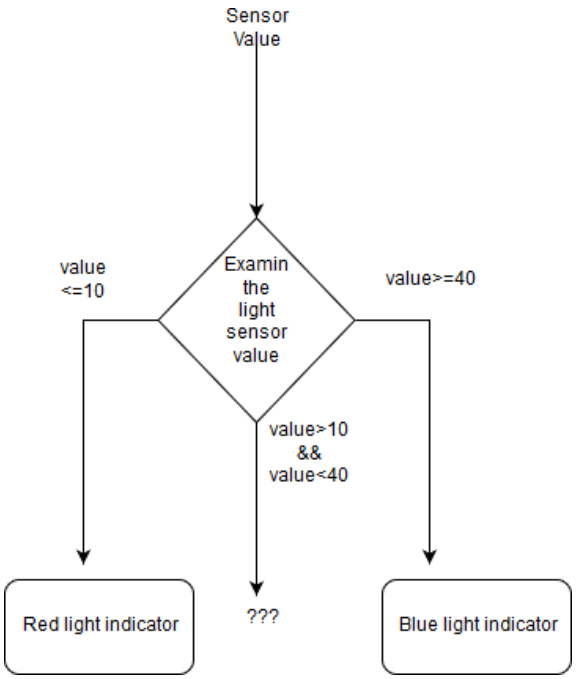

Figure 5.2.: Decision tree of logical requirement

In order to make the complex logical requirement into straightforward and clear requirement statement, one can put parenthesis (unique approach) or divide the requirement into number of statements (efficient approach)

5.7.2. Inverse Requirements

Negative requirements are problematic for requirement analyst. These sort of requirements describe what the system will not do. It is needed to transform the inverse requirements into positive manner as the negative requirements can be validated by the positive ones [26]. Example of an inverse requirement can be found in appendix Example 3.

Table 5.1 shows the inverse requirements at one side and recast requirements in positive sense on the other side. In order to eliminated the ambiguity aspect, negative requirements are often changed from passive voice into active voice.

Table 5.1.: Eliminating negation from functional requirements

Negative Requirements	**After rewriting (Positive sense)**
All sensors with two or more negative values should not be used	The system shall use only sensors having less than two values
The HMI interface will not have the capability to alter the car movement	Only steering shall be capable of altering the car movement
The warning signal will not be indicated when the door state is close	The system shall indicate warning signal when the door state is open

5.7.3. Omissions

Sometimes requirement lacks a vital piece of information which makes it is difficult for every reader to interpret the requirement in the same sense. A requirement may show the action taken by the system without specifying the reason of that action [26], [51]. The example of such requirement is stated above.

"The system shall produce a test report and send it to the main server."

In the above given example it is not recognized what causes the system to generate test report. Considering the given example, it is also important to know how the system will react when no error occur during the processing. Once can expect following actions:

- Perform nothing.
- Show a specific message such as "no error found".
- Produce an empty report and share it to the user.

5.7.4. Boundary

When boundaries are represented in numerical span, they can cause ambiguity in a requirement. It may indicate about the incomplete requirement [26]. Consider the following example in which the boundary is specified in a numerical range.

"The system shall indicate a warning message when the coolant temperature is greater than 90 degrees."

The example stated above raise question for the developers such as what will be the behavior of system when the coolant temperature is less than or equal to 90 degree. This issue can be resolved when a requirement addresses the complete numerical range interval. A bad example of expressing boundary in requirement specification is stated in appendix Example 1.
Furthermore, one can specify the terms such as inclusive and exclusive to clearly indicate if the value lies in between the numerical or not. This is illustrated in the following example.

"The system shall show warning message for fuel level 0 to 10 liters, inclusive"

5.7.5. Avoid Ambiguous Terms

Since the use of synonyms make it difficult for the developers to have a straightforward understanding about the requirement, it is advised to use consistent terms in requirement statements instead of synonyms or near synonyms. It is necessary to

mention such terms in the common glossary.
Abbreviations such as i.e or e.g are sometimes not universally understood by the developers, it could be possible that readers misunderstand the abbreviation and interpret requirement wrong[26].

5.7.6. Summary

This chapter provided a general overview about writing effective requirements specifications considering the approach proposed by this thesis to generate test cases using formal representation of requirements. At the beginning, guidelines for documenting requirements specifications were defined and the advantages associated with high quality requirements process were stated. The attributes of requirement statement were also explained in detail. Furthermore, different labeling techniques were presented in order to assign unique identifier to each requirement statement.

In the middle phase of this chapter, a requirement template was defined which helps in removing the ambiguity of NL requirements. The requirement template depicted a controlled NL structure which can be processed easily by the NLP techniques. Additionally, requirement style elements such as perspective based requirements and hierarchical requirements were also expressed in detail. At the end of this chapter, different techniques were proposed in order to manage different kind of ambiguities. Furthermore, detail discussion about various kind of requirement ambiguities was conducted such as complex logic, inverse requirements, omissions, boundary ranges and ambiguous terms.

6. Implementation

The focal point of this chapter is to explain the implementation details of the proposed solution. At first, it will introduce the tools and programming language used during the realization of the solution. In the next stage, different phases of implementation will be discussed, from documenting NL requirements specifications till the method proposed for the generation of test cases. At the end, a summary of the executed solution will be discussed.

6.1. Implementation Tools

For the implementation of solution proposed in the previous chapter, different tools and technologies are required during each step of implementation. Following are the details about different tools and techniques employed during the course of this thesis.

6.1.1. MS Excel

Requirements specifications for implementation purpose were documented in MS Excel. Furthermore, every signal specified in the requirements specifications were extracted and defined in an spreadsheet. This sheet contains detailed information about various attributes of an individual signal [55].

6.1.2. NLTK with Python

Python, a strong programming language was used with outstanding application for handling linguistic data. Additionally, Natural Language Toolkit (NLTK) was used with Python in order to provide a platform to construct NL processing programs [56].

During the implementation of the thesis, NLTK was used with python for tasks such as parts of speech tagging and syntactic parsing of each sentence.

6.1.3. Stanford Parser

It is an NL parser which operates based on grammatical formation of a sentence, for example, which collection of verbs are in conjunction (e.g. phrases) and identify the subject and object of a particular verb. The formation of phrase tree output is represented in Graphical User Interface (GUI) [32].

In the perspective of this thesis, outputs from dependency parsing functions of Stanford Parser are obtained which exhibit the relation between different entities and the dependency of each word in a sentence with each other.

6.2. Implementation Overview

The systematic approach proposed in Chapter Concept is executed with the help of different tools across each stage of implementation. The flow diagram illustrated in the Figure 6.1 shows the general approach towards implementing the proposed concept and it also depicts which steps are manual, semi-automated and automated, respectively.

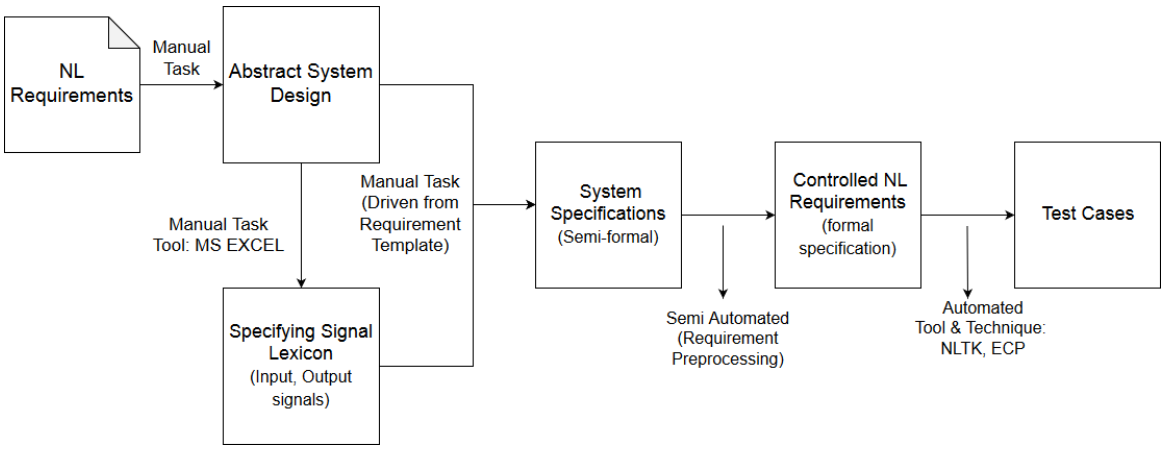

Figure 6.1.: Implementation Overview

6.3. Defining NL Requirements

This is the first step which involves the capturing of system requirements from system stakeholders. The requirements are expressed in informal textual form. These informal requirements are ambiguous, incomplete and not examined automatically. Therefore, the task will be to process them in the later stages of implementation and finally test the system based on formal representation of requirements.

It is to mention that example taken during the implementation of this thesis is from the Automotive domain. Following is the example of informal NL requirements documented from system stakeholders.

"At the beginning the system should check its initial state and after that it continuously monitors the fuel level and engine coolant temperature in a certain range interval respectively and if the fuel level value is very low or if the engine coolant temperature is very high then a warning signal will be indicated"

The information extracted from the informal requirements specifications can be expressed in the form of a use case as follows.

Precondition

The system has been initialized

Basic Flow

1. The fuel tank value is monitored in signed range interval values.
2. The engine coolant temperature value is monitored in unsigned range interval values.
3. The system validates if the fuel level is low.
4. The system validates if the engine coolant temperature is high.
5. The system validates by indicating a warning signal.

6.3.1. Abstract System Design

For the better conceptual understanding of the system it is necessary to represent it graphically. The abstract design helps the system stakeholders and developers to figure out the system.

Following figure displays the conceptual design of the NL requirements gathered during the initial phase of implementation. This design portrays the inputs taken by the system from various sources and output send to the HMI display.

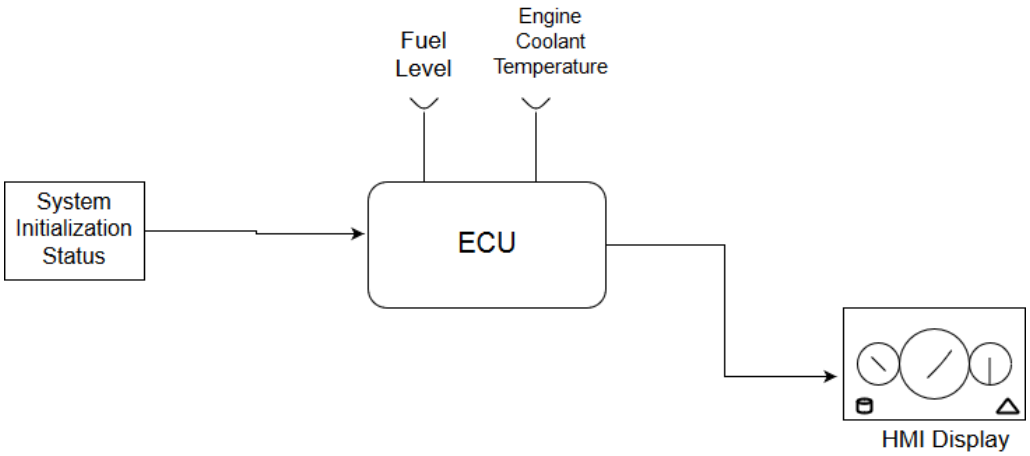

Figure 6.2.: Conceptual system design

6.3.2. Specifying Signal Lexicon for System

After portraying the system graphically, the follow up task is to define each input and output signal for the proposed system. For this purpose a signal lexicon document is created manually using MS Excel tool which includes detail information related to each signal such as data type, behavior at certain states, replacement value, sender and receiver of the signal in a spreadsheet.

Input Signals

1. To check if the system is initialized or not, signal such as Terminal-15 is used which indicates switched positive. If it is not connected or failed to connect, the rest of the system will not be initiated. This signal is received from Automotive Gateway (GW).

2. The input signal s_Fuel_Level is obtained from the fuel tank sensor. This signal measures the fuel level of the vehicle.

3. The input signal s_EngineCoolantTemperature is obtained from the coolant temperature sensor. It measures the engine coolant temperature of the vehicle.

	SignalName	Type	Algorithm	Remarks	Behavior at Init	Sender
2	s_Terminal15	unsigned char	253 - Init 254 - Replacement Value 255 - Error Value	Terminal 15	Error	GW
3	s_Fuel_Value	unsigned char	253 - Init 254 - Replacement Value 255 - Error Value	Car Fuel Value	Replacement Valu	Sensor
4	s_EngineCoolantTemperature	signed char	125 - Init 126 - Replacement Value 127 - Error Value	Engine Coolant Temperature	Replacement Valu	Sensor

Figure 6.3.: Signal Lexicon for input signals

Output Signals

1. The HMI_InfoLight output signal is used to indicate user on board by blinking different light signals. The output represents light warning based on the fuel level or engine coolant temperature value.

2. The HMI_InfoMessage output signal is used to indicate user on board by displaying various messages based on fuel level or engine coolant temperature value.

1	SignalName	Type	Algorithm	Remarks	Behavior at Init	Sender
2	HMI_InfoLight	unsigned char	0- Red 1 - Yellow 2 Green 253 - Init 254 - Replacement Value	Warning Light	Error	ECU
3	HMI_InfoMessage	unsigned char	0 - Warning Fuel Level is equal or below 10 Liter 1 - Warning Fuel is nearly finished 3 - ECT is equal or above 40 degree 253 - Init 254 - Replacement Value	Warning Message	Error	ECU

Figure 6.4.: Signal Lexicon for output signals

6.3.3. Extract System Requirements Specifications

The domain related information of specific terms are stated in the signal lexicon document. The terms specified in signal lexicon document are defined manually in the requirements statements such as signal name and their range interval. Each requirement statement must be represented by a labeling scheme. A single statement shall contain only one requirement. In order to handle the problems of ambiguity, inconsistency and incomplete requirements, various techniques are discussed in Chapter 5.

In the context of this thesis, a requirement template is suggested in Section 5.5 to handle the problems associated with NL requirements so that it can be easily processed by various NLP techniques. The derived requirements are represented in the list below.

1. The system shall check whether the initialization status value is valid.
2. The system shall take fuel level value from interval [0, 50].
3. The system shall take engine coolant temperature value from interval [-10, 110].
4. The system must display warning if the fuel level value is less than or equal to 10 liters.
5. The system must display warning if the engine coolant temperature value is greater than or equal to 90 degrees.

By now, the requirements are extracted based on the conceptual system design and the signal lexicon for system specification. Furthermore, controlled requirements are document based on a proposed requirement template. Following figure shows an example of template driven requirement statement.

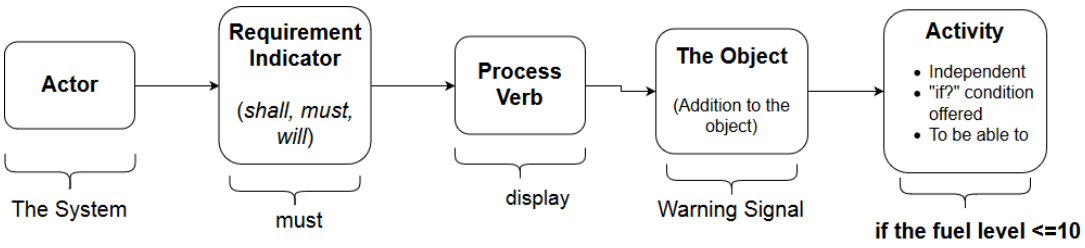

Figure 6.5.: Template driven requirement statement

6.3.4. Requirements Pre-processing

After the extraction of system specifications the requirements statements are normalized in which unwanted word are taken out and stemming is performed. The details about requirement pre-processing are described in section 5.4.2 of chapter 5. The output of this step is refined requirements specifications which can be easily processed by the NLP techniques.The normalized requirements specifications of the given system are as follows:

1. System shall check whether initialization-status-value is valid.
2. System shall take fuel-level-value from interval [0 , 50].
3. System shall take engine-coolant-temperature-value from interval [-10 , 110].
4. System must display warning-signal if fuel-level-value from interval [0 , 10].
5. System must display warning-signal if engine-coolant-temperature-value from interval [90 , 110].

6.3.5. Processing Requirements Specifications

In the previous section, the requirements specifications are represented in a formal manner. After expressing the requirements in a structure way, various NLP approaches are used in order to understand the text. These approaches include POS tagging, syntactical parsing, and dependency parsing methods. The NLTK platform is used to construct python programs for different NLP techniques.

POS Tagging

The POS tagging is performed in two steps.

1. Each word in the requirement statement is tokenized.
2. Individual tokens are assigned with parts of speech tags.

However, it is important to note that tokens include different words, numbers and symbols of a statement. This is shown in the following python code snippet.

```
>>> import nltk
>>>
>>> token = nltk.word_tokenize("System shall take fuel-level-value in interval [ 0, 50 ]")
>>> print("Parts of Speech:",nltk.pos_tag(token))
Parts of Speech: [('System', 'NNP'), ('shall', 'MD'), ('take', 'VB'), ('fuel-level-value', 'NN'), ('in', 'IN'),
 'CD'), (',', ','), ('50', 'CD'), (']', 'NN')]
```

Figure 6.6.: POS tagging NLTK function

In context of the requirements specifications of the system under consideration, the POS tagging for individual requirements specification are depicted graphically. The tagging information about the first requirement of the proposed system is shown below. This requirement contains the information about the systems precondition.

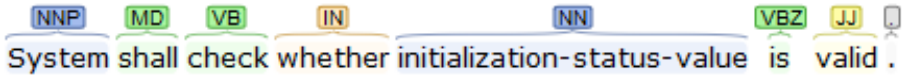

Figure 6.7.: Requirement Specification 1

The tagging information about the second requirement of the system is depicted below. It shows the input information about the fuel level. The fuel level is represented in a positive range interval [0,50]. The POS tagger also recognizes the parenthesis and numerical values.

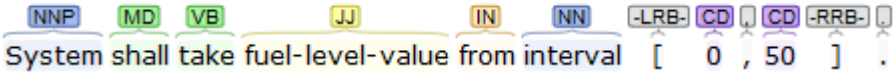

Figure 6.8.: Requirement Specification 2

Following figure represents the tagging details about the third requirement specification of the proposed system. It shows the input information about the engine coolant temperature (ECT). The ECT value is represented in unsigned range interval [-10, 110].

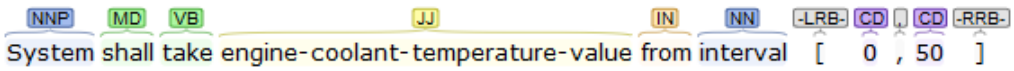

Figure 6.9.: Requirement Specification 3

The fourth and fifth requirements specifications show how the system behave if a value lies in a particular interval. The difference in these requirement statements is that they contain a conditional phrase. After applying the POS tagging, individual words are assigned with following labels. This is represented in the following figures.

- System/NNP must/MD display/VB warning-signal/JJ if/IN fuel-level-value/JJ from/IN interval/NN [/LSB 0/CD ,/, 10/CD]/RRB ./.
- System/NNP must/MD display/VB warning-signal/JJ if/IN engine-coolant-temperature-value/JJ from/IN interval/NN [/LSB 90/CD ,/, 110/CD]/RRB ./.

Syntactic Parsing

The requirement statements are parsed after the POS tagging. The output of parsing is syntactic tree structure which interpret the statement.

Parsing is perform in the following steps:

- In the first step, the POS tags identified from POS tagging approach are stated in the form of a sentence.
- In this step, a pattern is defined based on the regular expression.
- Regex parser is applied on the pattern.
- A chuck parser is created based on the given pattern.
- The results can be illustrated using print () or draw() function.

Following is the code snipped based on the template defined in order to parse the requirement statements of the proposed system.

```
>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked","VBD"),
>> pattern = """NP: {<DT>?<JJ>*<NN>}
.. VBD: {<VBD>}
.. IN: {<IN>}"""
>> NPChunker = nltk.RegexpParser(pattern)
>> result = NPChunker.parse(sentence)
>> result.draw()
```

Figure 6.10.: Parsing based on regex expression

Considering the above given approach following are the results of parsing obtain during the course of implementation of this thesis. In the context of test case generation it is vital to understand the parsing of an interval by a statement parser. Following is the list of elements recognized by the parser in a range:

1. Find the -LRB- and comparable -RRB-.
2. Join all the leaves between -LRB- and -RRB-

Following is the result obtained after the parsing of first requirement statement of the proposed system.

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD shall)
      (VP (VB check)
        (SBAR (IN whether)
          (S
            (NP (NN initialization-status-value))
            (VP (VBZ is)
              (ADJP (JJ valid)))))))
    (. .)))
```

Figure 6.11.: Parse result: Requirement Statement 1

After applying the parsing approach on the input requirement specifications such as requirement statement two and three of the system under consideration, following are the outcomes after applying parsing.

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD shall)
      (VP (VB take)
        (NP (NN fuel-level-value))
        (PP (IN from)
          (NP
            (ADJP (JJ interval)
              (NP (NNP -LSB-) (CD 0) (, ,) (CD 50)))
            (NNS -RSB-)))))
    (. .)))
```

Figure 6.12.: Parse result: Requirement Statement 2

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD shall)
      (VP (VB take)
        (NP (JJ engine-coolant-temperature) (NN value))
        (PP (IN from)
          (NP
            (ADJP (JJ interval)
              (NP (NNP -LSB-) (CD -10) (, ,) (CD 110)))
            (NNS -RSB-)))))
    (. .)))
```

Figure 6.13.: Parse result: Requirement Statement 3

The output statements of proposed system such as statement four and fifth have a conditional structures which are based on range interval such as [0, 10] and [90, 110]. The parser efficiently interpret such conditions if defined in a certain format. Following examples represent the result obtained after parsing the requirements statements four and five of the system suggested.

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD must)
      (VP (VB display)
        (NP (NN warning-signal))
        (SBAR (IN if)
          (FRAG
            (ADJP (JJ fuel-level-value)
              (PP (IN from)
                (NP
                  (ADJP (JJ interval)
                    (NP (NNP -LSB-) (CD 0) (, ,) (CD 10)))
                  (NNS -RSB-))))))))
    (. .)))
```

Figure 6.14.: Parse result: Requirement Statement 4

```
(ROOT
  (S
    (NP (NNP System))
    (VP (MD must)
      (VP (VB display)
        (NP (NN warning-signal))
        (SBAR (IN if)
          (FRAG
            (ADJP (JJ engine-coolant-temperature-value)
              (PP (IN from)
                (NP
                  (ADJP (JJ interval)
                    (NP (NNP -LSB-) (CD 90) (, ,) (CD 110)))
                  (NNS -RSB-))))))))
    (. .)))
```

Figure 6.15.: Parse result: Requirement Statement 5

6.3.6. Extracting Dependency Parsing

During the implementation phase, after parsing the requirement statements the task is to extract dependency of already parsed statement. This approach is used to find the association between head word and word which alter those heads[dependency parse]. For the purpose of extracting dependencies after paring the sentence, a tool called Stanford Parser was used. This tool helps in evaluating different kind dependencies such as basic dependency, enhanced dependency, and Open Information Extraction dependencies. However, during the course of this implementation, only the basic parse dependency method was applied.

In perspective to the solution proposed, dependency parsing is applied to each requirement statement of the system under consideration. After the evaluation of basic dependencies, each requirement will be represented in a conceptual model.

Dependency Parsing: Requirement Statement 1

The dependency parsing of requirement representing the systems precondition is shown below.

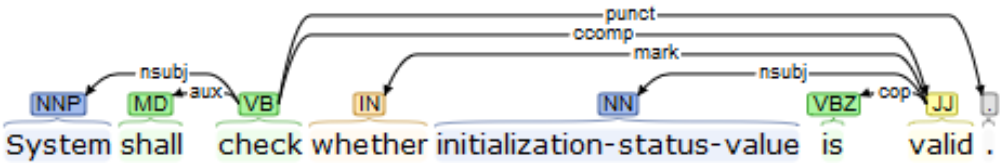

Figure 6.16.: Dependency Parsing: Requirement Statement 1

After extracting the basic dependencies, terms such as "system" defines the actor, whereas "check" annotates the action and at the end a condition is defined with constraint to prove if the initialization status is valid. Based on the dependencies we can represent the entities in conceptual form, as shown in the figure below.

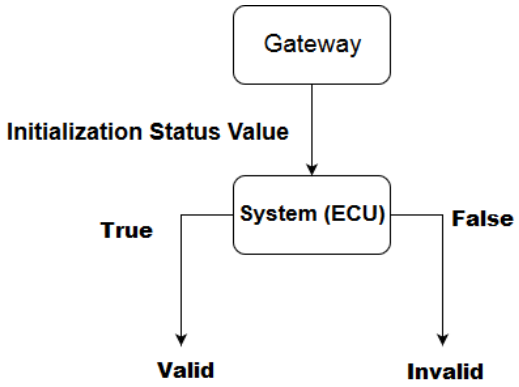

Figure 6.17.: Conceptual Model: Requirement Statement 1

Dependency Parsing: Requirement Statement 2

Following figure illustrates the dependency parsing of the input requirement statement 2 for the suggested system.

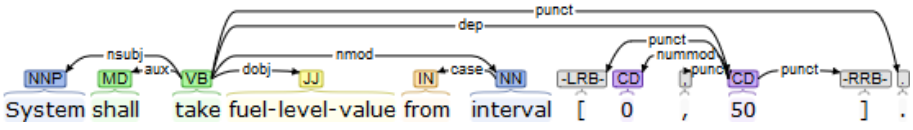

Figure 6.18.: Dependency Parsing: Requirement Statement 2

The abstract model driven from the basic dependencies is depicted in figure below.

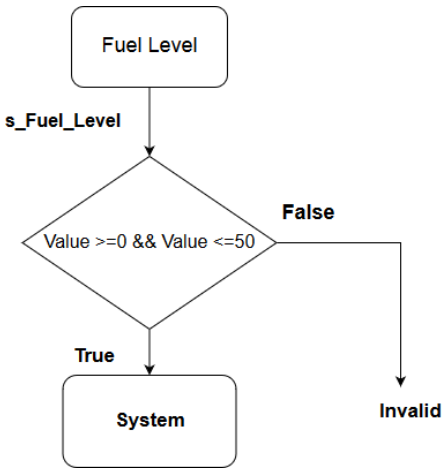

Figure 6.19.: Conceptual Model: Requirement Statement 2

Dependency Parsing: Requirement Statement 3

Following figure depicts the dependency parsing of the input requirement statement 3 for the proposed system.

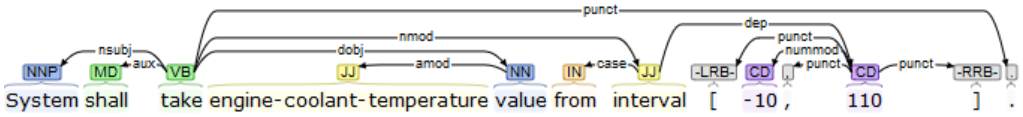

Figure 6.20.: Dependency Parsing: Requirement Statement 3

The conceptual model based basic dependency is depicted in figure below.

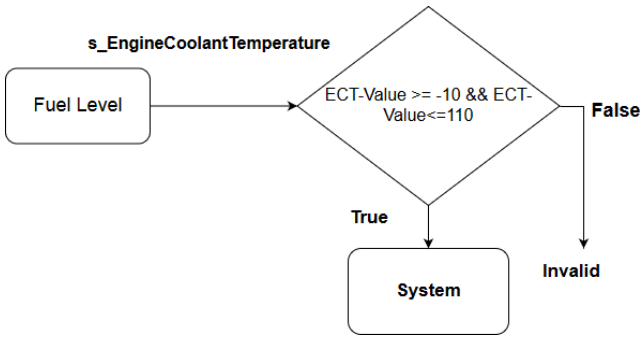

Figure 6.21.: Conceptual Model: Requirement Statement 3

Dependency Parsing: Requirement Statement 4

Following figure illustrates the dependency parsing of the output requirement statement 4 for the suggested system.

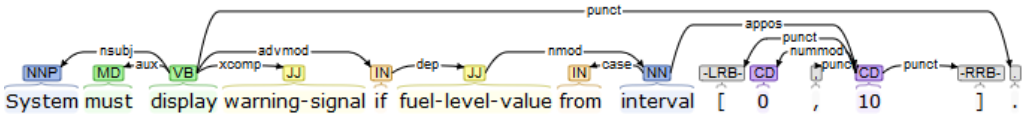

Figure 6.22.: Dependency Parsing: Requirement Statement 4

The abstract model driven from the basic dependencies is depicted in figure below.

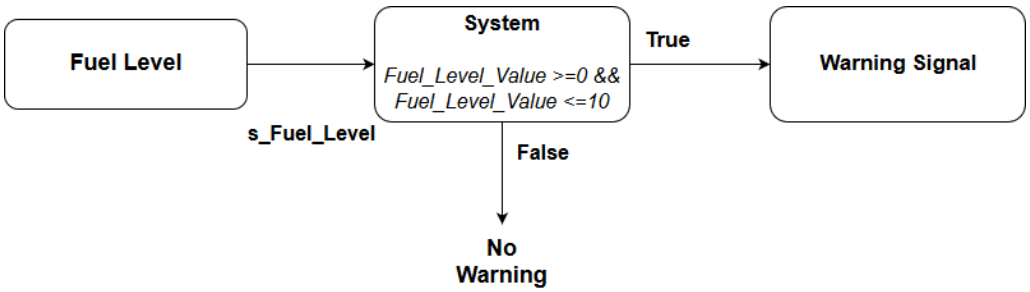

Figure 6.23.: Conceptual Model: Requirement Statement 4

Dependency Parsing: Requirement Statement 5

Following figure depicts the dependency parsing of the output requirement statement 5 for the proposed system.

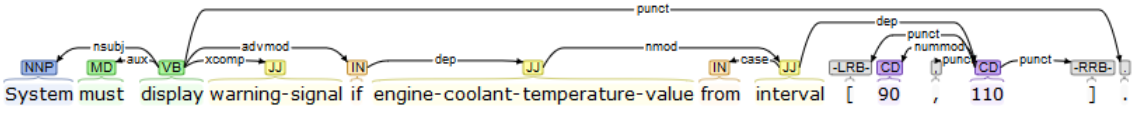

Figure 6.24.: Dependency Parsing: Requirement Statement 5

The conceptual model based dependency is depicted in figure below.

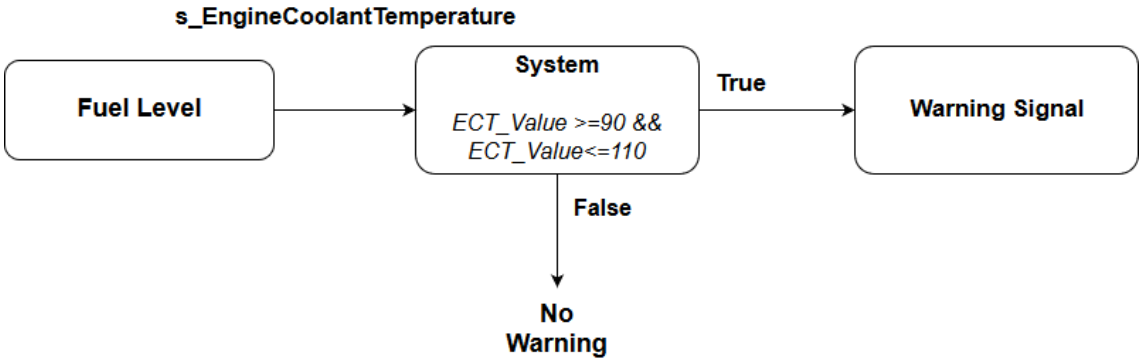

Figure 6.25.: Conceptual Model: Requirement Statement 5

6.3.7. Generating Test Cases

The input, processing state and output are identified based on the conceptual modeling representation after applying the dependency parsing technique. As the derived conceptual model illustrates the control flow of the system, therefore black box testing can be performed depending upon the values taken as input and how this input is processed by the system to produce an output respectively. The following Figure 6.26 illustrate the formal representation of system specification.

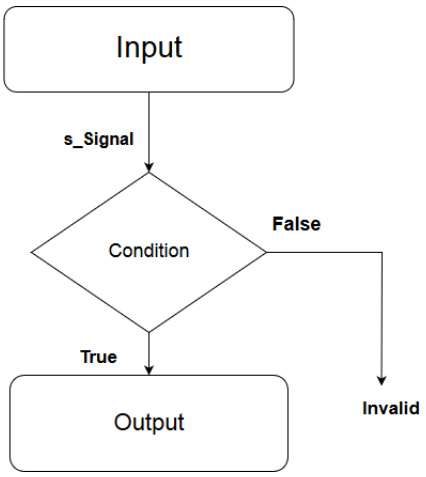

Figure 6.26.: Conceptual model as formal representation

In order to derive the test cases, a method which splits the input details of software units into divisions of equivalent data, this method is called Equivalence class partitioning (ECP) [46]. This method specifies the test cases which detect error classes. The big advantage of this approach is that it minimizes the number of test cases to be developed. Therefore reduces the time taken.

Consider an input value defined in a range interval [a, b]. We are required to cover the overflow in positive side, negative side or neither of these two, based on this factor, 3 partitions are created. The given range interval can be defined as

follows.

$$INT\_MIN \leq a + b \leq INT\_MAX$$

and

$$with \quad a \in \{INT\_MIN, ..., INT\_MAX\} \quad and \quad b \in \{INT\_MIN, ..., INT\_MAX\}$$

Whereas, considering the precise condition of equality the test vector values are $INT\_MIN = a + b$ and boundary values is denoted as $INT\_MAX = a + b$.

6.3.8. Summary

This chapter presents the execution of proposed solution. At first different tools and techniques used during the course of implementation were discussed. Tools and techniques such as NLTK with Python, Microsoft Excel and Standford Parser were used in a systematic manner, in order to find the solution for the problem. The V-model of software development was followed throughout the implementation process.

The objective of this thesis is to formalize the representation of system specifications from natural language requirements. In the first step, requirements were documented from system stakeholders in NL form. In the following step, a conceptual system design was presented based on stakeholders specifications which expressed the basic flow of the system. Afterwards, input and output signals were defined in detail. Using the detailed description of signals, requirements statements were documented in a proposed template. At the end, requirements statements were refined and various NLP techniques were applied for extracting information from each requirement statement. Furthermore, a method was proposed to generate test cases based on template requirements.

7. Evaluation

This chapter describes the observations gained during the implementation of the proposed solution. Moreover, the challenges faced during the systematic execution of individual tasks will also be discussed. Next, a comparison of recommended solution with the previously existing approaches will be done.

7.1. Formalizing NL Requirements

7.1.1. Objectives

As it has been discussed in the previous chapters that NL requirements have problems such as unclear meaning, inconsistency and incompleteness. Therefore a system based on these requirements is not verifiable.

The focus of proposed approach is to represent the requirements in a structured manner in order to eradicate the potential problems associated with them. Followings were the aims specified while formalizing the NL requirements specifications.

- A large set of NL requirements usually represented in a paragraph. The aim was to identify and express them in individual statements.
- Each individual requirement should be represented in a specific or controlled structure.
- A technique should be defined in order to represent system requirement specifications. Therefore, documenting requirements in a superlative manner require restricted grammar, correct spelling and a logical cooperation in a sentence.
- Define the requirement structure based on either user perspective or system perspective.

7.1.2. Achievements

Following are the objectives successfully implemented based on the proposed solution in the chatper 5.

- The requirements from system stakeholders were identified and converted into discrete requirement statements. The order of requirements was maintained the same as described in the stakeholders statements.

- For the purpose of requirement traceability and refinement, a distinctive identifier is assigned to individual requirements. During the implementation phase, a sequence numbering technique was presented which marks the start of requirement with a distinctive number.
- To tackle the ambiguous nature of informal requirements and their non deterministic interpretation, a requirement template was proposed. Therefore, a controlled structure for each requirement statement was realized.
- Different requirement ambiguities were managed based on the following techniques.
  - Requirement with complex logic was divided into number of statements. Those statements were labelled with sequence numbering.
  - The negative requirements were rewritten and described in a positive sense.
  - In order to handle the ambiguities of numerical range, a range interval was specified stating the minimum and maximum boundary.
  - As the synonyms and near synonyms cause ambiguity, therefore consistent terms were defined and used repeatedly in each requirement statement.

7.1.3. Results and Discussion

Since a systematic approach is followed in terms of formalizing the NL requirements, therefore various observations are gained during each step. The transformation of informal textual requirements using requirement templates was performed manually as the informal requirements are highly ambiguous and inconsistent, therefore careful elicitation of requirements was required. Following are the key information extracted in a specific sequence during the requirement elicitation process.

1. Precondition (Initial state)
2. Basic flow
   a) Input step
   b) Conditional step
   c) Output step
3. Post Condition (Optional)

The requirement statements produced during the requirement collection phase are semi formal with potential ambiguities. Following points were considered in order to express the requirements in a deterministic and formal manner.

- Represent the requirements in active voice.

- Avoid the use of articles such as "The", "A", "An" in a requirement sentence.
- Concatenate the Adjectives with their respective Nouns.
- Avoid using inverse requirements.
- Always specify the maximum and minimum of a boundary value.

Additionally, a requirement template was proposed which formalizes the sentence and represent it in a controlled structure. The following figure shows the transformation of semi formal requirement to formal requirement specification driven from the rules given above and proposed requirement template. The formal representation generated at the end is then applied for natural language processing techniques.

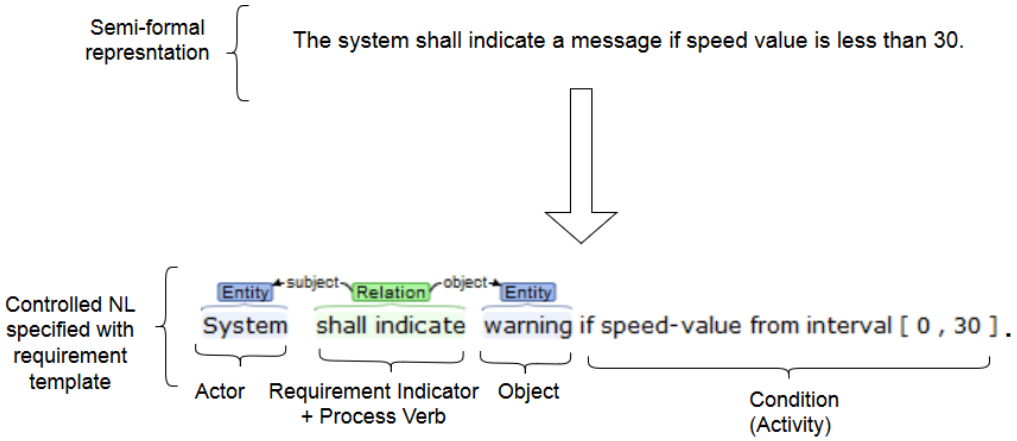

Figure 7.1.: Test constraints requirement statement 5

7.1.4. Issues Observed

The user requirements elicitation from the system stakeholders is a time consuming activity as it require careful capturing of the requirements. The informal requirements are syntactically ambiguous and semantically inconsistent. Semi formal requirements are produced from informal textual requirements after the requirement elicitation process. These requirements are highly ambiguous, thus those can not be processed by the machine. The rules specified in order to deal with the ambiguity are very rigorous and limited, therefore as a result a controlled and restricted structure (Requirement template) is proposed. This controlled structure can not define the complex and long requirement statements. Therefore, only short and direct requirement sentences are generated for test case generation.

7.2. Processing NL Requirements

7.2.1. Objectives

The informal requirements were transformed into controlled natural language form so that different processing techniques can be applied. Following are the goals specified during the requirement processing approach.

- Identify each term in a requirement sentence.
- Analyze each term in a requirement statement and perform a syntactic analysis in order to understand the statement.
- Perform a semantic analyzing which help in understanding the meaning of the requirement statement.
- Breakdown each statement based on the semantic analysis, and identify the entities and the relationship between them.

7.2.2. Achievements

The goals specified during the requirement processing phase were realized using the natural language processing techniques. Following are the different processing techniques used in order to satisfy the goals.

- Each term in a sentence was identified using part of speech (POS) tagging approach. Firstly, each word in a sentence were tokenized and afterwards, parts of speech tags were assigned to each token.
- Syntactic parsing was performed on the requirement statements with POS tags. Number of syntactic trees were produced as a result, which interpreted the sentences.
- Dependencies between each word in a sentence were extracted using the dependency parsing technique. In this method the association between heads words and words which alter those heads were recognized.
- Based on the dependency parsing, a conceptual model was designed which represents the entities and the relationship between them. Furthermore, it has also provided the basic flow and the constraints require in order to generate the test cases.

7.2.3. Results and Discussion

The NLP approaches depend on the structure of a sentence therefore, it was necessary to convert semi formal requirements into a controlled structure. In order to illustrate this, the author had taken two sentences with, one of them represents the

semi formal requirement while the other is driven from requirement template. The comparison by applying POS tagging is illustrated as follows.

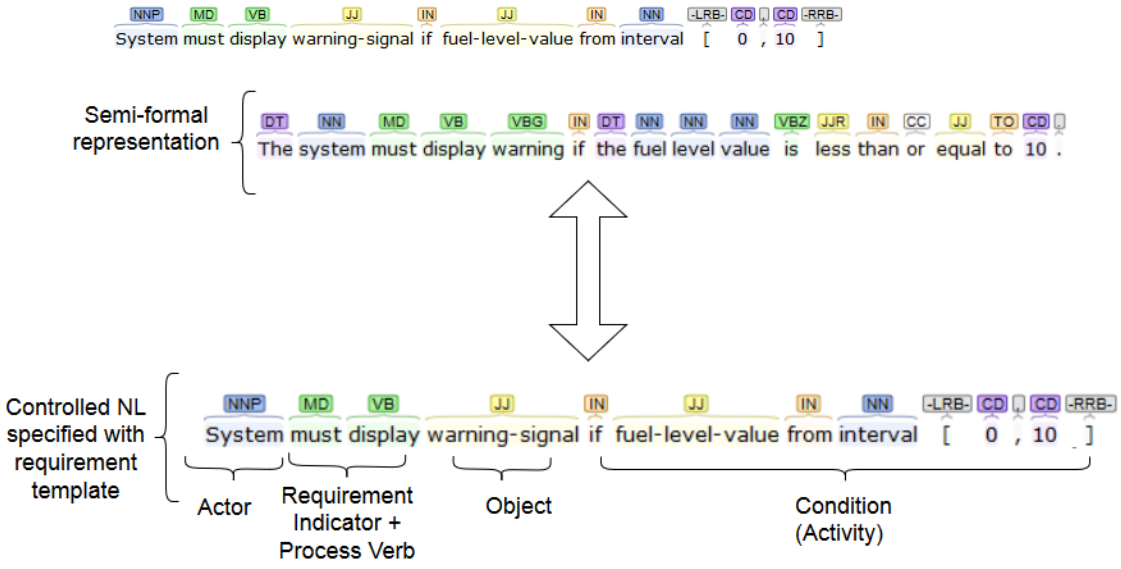

Figure 7.2.: POS Tagging: Comparing semi-formal requirement with controlled NL

From the above example it can be seen that semi formal requirements use extra tags which may result in the misinterpretation of the requirement while processing. Therefore, a consistent and deterministic solution was proposed based on the controlled natural language (CNL).
Following is the comparison of results drawn after the dependency parsing, as it can be seen that semi-formal requirements has more structural complexity as compare to the requirements expressed in controlled NL.

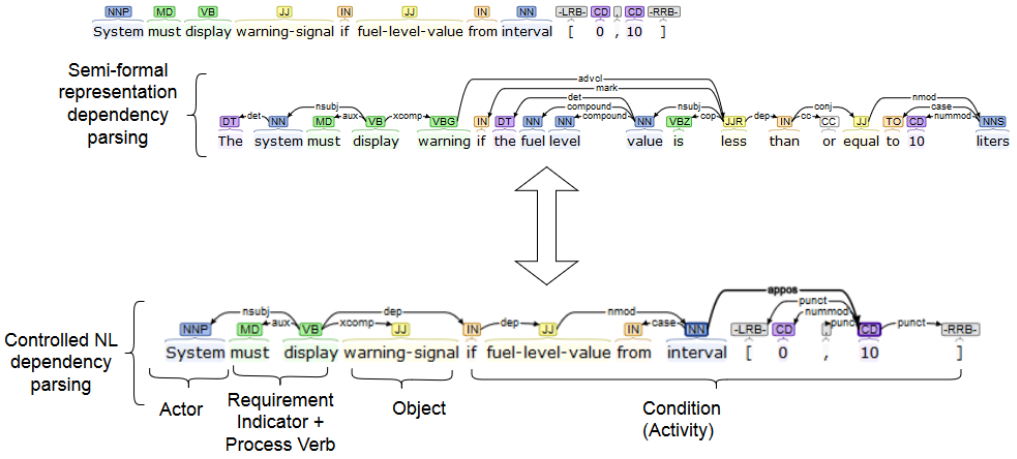

Figure 7.3.: Comparing structural complexity

The complexity of syntactical structures is evaluated using structural complexity (SC). The SC of a particular sentence is the total length of dependency links in the structure [57]. For the sake of understanding, consider two sentences S1 and S2, they have same number of words and have identical meaning, if SC(S1) >SC(S2) then sentence S1 is hard to process as compared to sentence S2.

After calculating the structural complexity for both semi-formal representation and controlled NL, following results were obtained.

- Structural complexity for semi formal sentence is 37 (total length of dependency links)
- Structural complexity for controlled NL sentence is 21
- The SC difference calculated between the given semi-formal sentence and controlled NL sentence is actually the difference of total length of dependency links in the statements, which is 37 - 21 = 16

Hence, by comparing the results the structural complexity is reduced to 56.7% by expressing the requirements in controlled natural NL.

It has been evaluated that representing the requirements in a controlled structure reduces the structural complexity. In order to illustrate the case, requirements from proposed system are taken into consideration. It is important to note that full stop (".") is not taken into contemplation as it effects the structural complexity to a great extent. Following figure depicts the reduced structural complexity for the requirements used in this thesis when compared with semi formal requirements.

Figure 7.4.: Comparing structural complexity

Following observations are obtained from the graph given above:

- Requirement 1: structural complexity difference is 26 - 13 = 13 and the results are improved to 50% using controlled NL.
- Requirement 2: structural complexity difference is 27 - 22 = 5 and the results are improved to 18.52% using controlled NL.

- Requirement 3: structural complexity difference is 32 - 22 = 10 and the results are improved to 31.25% using controlled NL.
- Requirement 4: structural complexity difference is 37 - 21 = 16 and the results are improved to 38.23% using controlled NL.
- Requirement 5: structural complexity difference is 37 - 21 = 16 and the results are improved to 38.23% using controlled NL.

7.2.4. Issues Observed

As the requirements were documented and processed in natural language, it was compulsory to strictly follow the grammatical rules. During the course of implementation it was observed, if a requirement expressed in a false grammar or had spelling mistakes then NL processing techniques failed to interpret such a statement. Furthermore, incorrect interpretation of NL requirements could further affect the subsequent process. Various limitations were noticed while following the requirement templates such as concatenation of adjectives with its nouns. Although this approach reduces the structural complexity of a sentence but still the NLP techniques frequently misinterpret the noun as adjective. In order to carefully handle the rigorous aspects of controlled NL language human monitoring is required.

7.3. Requirement Based Testing

7.3.1. Objectives

After making the NL requirements syntactically deterministic and semantically consistent the follow up task was to generate the test cases based on those requirements. In this context following goals were defined in order to produce test cases driven from requirement templates.

- Identify the entities and signals mentioned in the controlled NL requirements.
- Propose a black box testing technique that can efficiently and effectively handle range intervals
- Specify test constraints that comprehensively cover the given range interval

7.3.2. Achievements

Following are the aims successfully executed based on the proposed solution.

- Using the dependency parsing approach, entities were recognized and a basic flow was defined with the help of an abstract model. This model defined the actors, their action and the object. Furthermore, the inputs and output signals were also defined.

- A black box testing technique called equivalence class partitioning was proposed in order to intelligently analyze and handle the range intervals.
- Test scenarios were defined based on the partitions driven from the equivalence class partition approach.

7.3.3. Test Case Generation from Processed Requirements

Since the manual software testing is a labour intensive and error prone activity. Therefore, it is practically difficult to perform testing for individual set of test data, particularly if there exist sizeable amount of input combinations.

The ultimate goal of this thesis is to generate test cases based on formal representation of system specifications. In this regard, we have searched for special approaches that can cleverly cover the test cases from a huge pool of test scenarios. In terms of accomplishing the goals, technique such as Equivalence Class partitioning is taken into consideration.

Following is the application of equivalence class partitioning approach in order generate of test cases based on system requirement specification.

Testing Requirement Statement 1

The first requirement specification is the proposed precondition for the system under consideration, it is stated as follows.

"System shall check whether initialization-status-value is valid."

After applying the NLP techniques such as POS tagging, sentence parsing and extracting the dependencies between words, it was evaluated that the test cases for the given requirement statement are based on certain values.

The signal such as s_Fuel_Level(Initialization-status-value) mentioned in the signal lexicon document represents different states such as "init", "replacement value" and "error value", with values such as 253, 0 and 255 respectively. Following is the algorithmic illustration of test scenarios.

```
if(Initialization_Status_Value == 253)
{
// System validates the initialization status
}
else if(Initialization_Status_Value == 0)  // Executed as replacement value
{
 //Executed if no value is received by the system (replacement value)
}
else
{
// System indicates the invalid initialization status value
}
```

Figure 7.5.: Test constraints requirement statement 1

Testing Requirement Statement 2

The requirement statement two describes that the system should obtain fuel level values in a range interval [0, 50], for this purpose different constraints are defined. The fuel level value is represented in data type unsigned char with value range from 0 to 255.

In order to generate test cases that covers the whole range, technique such as equivalence class partitioning is applied. This is portrayed in the following picture in which four partitions are shown. Partition 1 and partition 3 defines invalid values, other than that range for the valid fuel level lies in the partition 2, whereas the algorithmic values defined in the signal lexicon file shows the values such as init, replacement value, and error values

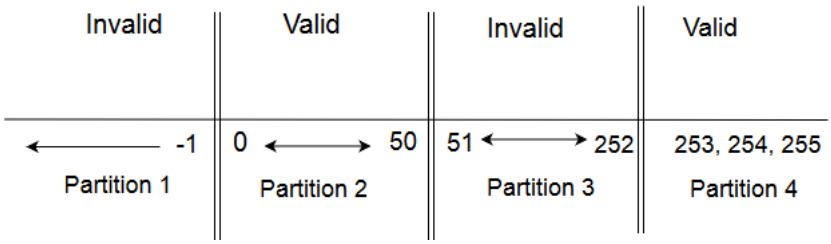

Figure 7.6.: Class partitioning for fuel level range

Based on the partitioning, an algorithm can extract the test scenarios comprehensively. This is illustrated in the code snippet below.

```
if(fuel_level_value >= 0 && fuel_level_value <=50)
{
// Valid fuel level value
}
else if(fuel_level_value == 253)
{
// Init value
}
else if(fuel_level_value == 254)
{
// Replacement Value
}
else if(fuel_level_value == 255)
{
// Error value
}
else {
//Invalid Value
}
```

Figure 7.7.: Test constraints requirement statement 2

Testing Requirement Statement 3

In this requirement statement it is stated that the system accepts engine coolant temperature values in a range interval [-10, 110]. The data type defined for engine coolant temperature is signed char. Following figure shows the different partitions derived for this range interval.

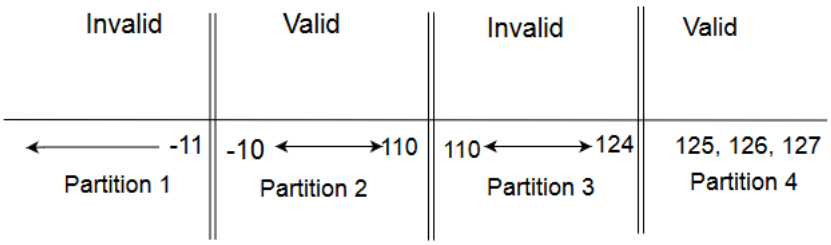

Figure 7.8.: Class partitioning for engine coolant range

In order to generate the test cases for the engine coolant temperature value. The code obtained after applying equivalence class partitioning is illustrated as follows.

```
if(engine_coolant_temperature_value >= -10 && engine_coolant_temperature_value <= 110)
{
// Valid engine coolant temperature value
}
else if(engine_coolant_temperature_value == 123)
{
// Init value
}
else if(engine_coolant_temperature_value == 124)
{
// Replacement Value
}
else if(engine_coolant_temperature_value == 125)
{
// Error value
}
else {
//Invalid Value
}
```

Figure 7.9.: Test constraints requirement statement 3

Testing Requirement Statement 4

This requirement statement indicates an warning signal if fuel level value lies in a range interval [0, 10]. The class partitioning for a fuel range is depicted as follows.

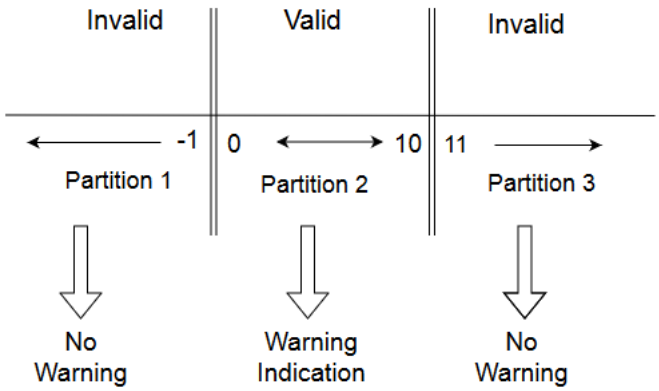

Figure 7.10.: Correlation between fuel level and warning signal

Following code snippet shows a warning signal is triggered if fuel value falls in the defined range interval. The signal value "1" indicates that warning is turned on and value "1" designates that warning signal is turned off.

```
if(fuel_level_value >= 0 && fuel_level_value <= 10)
{
 Warning_signal= 1
}
else
{
 Warning_signal= 0
}
```

Figure 7.11.: Test constraints requirement statement 4

Testing Requirement Statement 5

This requirement statement shows a warning signal, if engine coolant temperature value falls in a interval [90, 110]. The class partitioning for engine coolant temperature is depicted as follows.

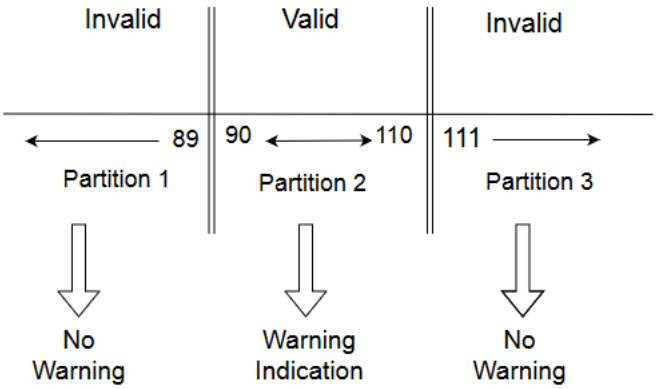

Figure 7.12.: Correlation between engine coolant temperature and warning signal

Following code snippet illustrates test scenarios how a warning signal is displayed if fuel value lies down in the defined range interval. The signal value "1" shows that warning is turned on and value "0" designates that warning signal is turned off.

```
if(engine_coolant_temperature_value >= 90 && engine_coolant_temperature_value <= 110
{
 Warning_signal= 1
}
else
{
 Warning_signal= 0
}
```

Figure 7.13.: Test constraints requirement statement 5

7.3.4. Results and Discussion

A black box testing method called equivalence class partitioning (ECP) was proposed to test the range interval specified in a requirement. This testing techniques is applicable to all levels of testing such as unit, integration, system and acceptance testing. Using this approach, different partitions were created. Each partition contained a particular range of numerical values.

After applying equivalence class partitioning method to the requirements specifications containing range intervals, following results were obtained.

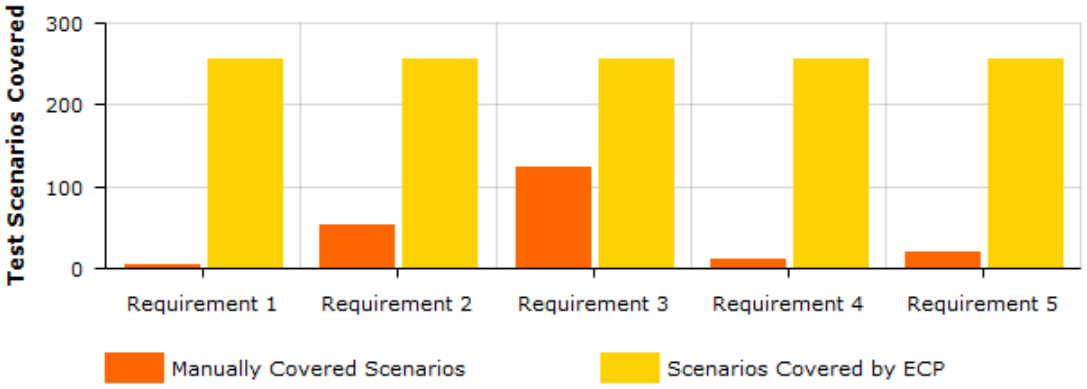

Figure 7.14.: Comparing covered test scenarios

Considering the above given graph a comparison of manually created test scenarios with ECP generated test cases was presented. Furthermore, the manually created test scenarios are covered in a particular time frame. This comparison is illustrated in the following points.

- Requirement 1: the manually created test cases were approximately 4 out of 256 possible test cases, hence only covering the 1.56% of the total range. Whereas applying ECP always covers the overall range.
- Requirement 2: the manually created test cases were approximately 54 out of 256 possible test cases, hence only covering the 26.09% of the total range. Whereas applying ECP always covers the overall range.
- Requirement 3: the self created test scenarios were approximately 124 out of 256 possible test cases, hence only cover 48.43% of the whole range.
- Requirement 4: the manually created test cases were approximately 11 out of 256 possible test cases, hence only covering the 4.29% of the total range.
- Requirement 5: the manually created test cases were approximately 20 out of 256 possible test cases, hence only covering the 7.81% of the total range.

7.4. Summary

In this chapter, the assessments were obtained from the activities such as formalizing NL requirements, processing NL requirements, and requirement based testing. During the analysis of each activity specified in this chapter, the author compared the achievements obtained with the goals defined at the beginning. Furthermore, results obtained after implementing the proposed solution were graphically illustrated and compared with the results captured from past approaches. Finally, the issues faced during the course of implementation were discussed in detail.

8. Conclusion

This chapter sums up the work that has been performed during the research. Following are the outlines of challenges that the author has observed and future work to improve the suggested solution.

8.1. Challenges

Documenting NL requirements specifications from system stakeholders is a manual activity, therefore it is time consuming. Considering the aspect that requirements are elicited in NL form and then based on requirements template a specific structure is defined. Controlled NL requirements are driven from requirement template, therefore require careful handling of requirements. It was also observed during the implementation phase, requirement template was always considered as a reference but due to the restricted nature of requirement template, it makes the formal representation of system specifications a rigorous process. Furthermore, controlled structure of requirement templates do not assist in defining complex and long requirement statements. Therefore, only short and direct requirement sentences are generated for test case generation.

During the research, it was observed that the key focus is towards manual ambiguity detection while avoiding or fixing ambiguity automatically was largely neglected. The author has also experienced that for the correct interpretation of requirements, it was necessary to write the requirements with correct grammatical rule. During the implementation phase, it was observed that requirements with false grammar or spelling mistakes were not interpreted by the NLP techniques. Other than that, in order to interpret the requirements during the implementation phase various constraints were defined such as concatenation of adjectives with its nouns. Although this approach reduced the structural complexity of a sentence but still the NLP techniques frequently misinterpret nouns as adjectives.

8.2. Future Work

The proposed requirement templates should be reconsidered and contemplate the domain-related ontologies to ease the restricted constraints and the vocabulary of the requirements statements [12]. With more improved requirement statements, the future work would be to take out the object oriented information from software requirements statements such as class, attributes, association, generalization etc. [16].

By automatically extracting this information, conceptual models with more details can be generated based on NL requirements specifications. This can further help us in generating more deterministic test cases. Since NL domain is vast, therefore comprehensive coverage of compound requirements statements is not practicable. This would not happen over night. Thus we require a tool, in order to automate this practice. Furthermore, a database is needed for saving the graph produced [58].

8.3. Concluding Remarks

This research introduces the requirement template in order to generate formal representation of system specifications from NL requirements statements. After representing requirements statements in a template, controlled NL requirements specifications are obtained. Problems of ambiguous requirements are resolved by representing NL requirements in a controlled structure. The proposed systematic approach has interpreted the controlled NL requirements using NLP techniques. Conceptual models are derived from controlled NL using the interpretation from NLP approaches. Finally, test constraints are created based on the range interval partitioning method. Thus, it has been proved that test cases can be generated from controlled NL requirements specification. Although this approach is partially automated but with some future improvements (as discussed in the previous Section 8.2.) we can achieve a fully automated approach for the generation of test case from controlled NL requirements.

Bibliography

[1] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, 2004.

[2] Bertrand Meyer. On formalism in specifications. In *Program Verification*, pages 155–189. Springer, 1993.

[3] Daniel Popescu, Spencer Rugaber, Nenad Medvidovic, and Daniel M Berry. Reducing ambiguities in requirements specifications via automatically created object-oriented models. In *Monterey Workshop*, pages 103–124. Springer, 2007.

[4] Maximiliano Cristiá and Brian Plüss. Generating natural language descriptions of z test cases. In *Proceedings of the 6th International Natural Language Generation Conference*, pages 173–177. Association for Computational Linguistics, 2010.

[5] Clementine Nebut, Franck Fleurey, Yves Le Traon, and J-M Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[6] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[7] Antonia Bertolino. Software testing research and practice. In *International Workshop on Abstract State Machines*, pages 1–21. Springer, 2003.

[8] Theodore Hammer, Linda Rosenberg, Lenore Huffman, and Lawrence Hyatt. Measuring requirements testing:: experience report. In *Proceedings of the 19th international conference on Software engineering*, pages 372–379. ACM, 1997.

[9] Ram Chatterjee and Kalpana Johari. A prolific approach for automated generation of test cases from informal requirements. *ACM SIGSOFT Software Engineering Notes*, 35(5):1–11, 2010.

[10] Muthu Ramachandran. Requirements-driven software test: a process-oriented approach. *ACM SIGSOFT Software Engineering Notes*, 21(4):66–70, 1996.

[11] Vector Autosar Components. `https://www.vector.com/de/de/produkte/produkte-a-z/software/davinci-configurator-pro/,Accessed20.4.2019/`, 2019.

[12] Ashfa Umber and Imran Sarwar Bajwa. Minimizing ambiguity in natural language software requirements specification. In *2011 Sixth International Conference on Digital Information Management*, pages 102–107. IEEE, 2011.

[13] Brian Dunbar. Nasa - sts-51l mission profile. `https://www.nasa.gov/mission_pages/shuttle/shuttlemissions/archives/sts-51L.html/`, 2017.

[14] Douglas N. Arnold. The Explosion of the Ariane 5. `http://www-users.math.umn.edu/~arnold/disasters/ariane.html/`, 2000.

[15] L Mich, M Franch, and P Novi Inverardi. Requirements analysis using linguistic tools: Results of an on-line survey. *Requirements Engineering Journal*, 2, 2003.

[16] Noraini Ibrahim, Wan MN Wan Kadir, and Safaai Deris. Documenting requirements specifications using natural language requirements boilerplates. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 19–24. IEEE, 2014.

[17] Muhammad Touseef and Zahid Hussain Qaisar. A use case driven approach for system level testing. *arXiv preprint arXiv:1212.3060*, 2012.

[18] Luay Ho Tahat, Boris Vaysburg, Bogdan Korel, and Atef J Bader. Requirement-based automated black-box test generation. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 489–495. IEEE, 2001.

[19] Jon Holt. *UML for Systems Engineering: watching the wheels*, volume 4. IET, 2004.

[20] Ravi Prakash Verma and Md Rizwan Beg. Generation of test cases from software requirements using natural language processing. In *2013 6th International Conference on Emerging Trends in Engineering and Technology*, pages 140–147. IEEE, 2013.

[21] Klaus Pohl. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam-foundation level-IREB compliant*. Rocky Nook, Inc., 2016.

[22] Kevin Ryan. The role of natural language in requirements engineering. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 240–242. IEEE, 1993.

[23] Barry W Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE transactions on software engineering*, 14(10):1462–1477, 1988.

[24] Robert B Grady. An economic release decision model: Insights into software project management. *ASMC, Software Quality Engineering*, pages 227–239, 1999.

[25] Donald C Gause and Gerald M Weinberg. *Exploring requirements: quality before design*, volume 7. Dorset House New York, 1989.

[26] Karl Wiegers. *More about software requirements: thorny issues and practical advice*. Microsoft Press, 2005.

[27] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: formal object-oriented language for communicating systems*. Prentice Hall, 1997.

[28] P Gibson. Formal requirements models: Simulation, validation, and verification. *Department of Computer Science, Maynooth*, 2001.

[29] Constance Heitmeyer, James Kirby, and Bruce Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of COMPASS'97: 12th Annual Conference on Computer Assurance*, pages 35–47. IEEE, 1997.

[30] Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 10(5):56–62, 1997.

[31] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, pages 165–174. ACM, 2011.

[32] Dan Klein and Christopher Manning. Standford parser 1.6, standford natural language processing group. `https://nlp.stanford.edu/software/lex-parser.shtml/`, 2014.

[33] Peter Bollen. Sbvr: A fact-oriented omg standard. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 718–727. Springer, 2008.

[34] Sri Fatimah Tjong, Nasreddine Hallam, and Michael Hartley. Improving the quality of natural language requirements specifications through natural language requirements patterns. In *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, pages 199–199. IEEE, 2006.

[35] Noraini Ibrahim, Wan MN Wan Kadir, and Safaai Deris. Propagating requirement change into software high level designs towards resilient software evolution. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 347–354. IEEE, 2009.

[36] Elisabeth Métais, Farid Meziane, Mohamad Saraee, Vijayan Sugumaran, and Sunil Vadera. *Natural Language Processing and Information Systems: 21st International Conference on Applications of Natural Language to Information Systems, NLDB 2016, Salford, UK, June 22-24, 2016, Proceedings*, volume 9612. Springer, 2016.

[37] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2010.

[38] Lorijn van Rooijen, Frederik Simon Bäumer, Marie Christin Platenius, Michaela Geierhos, Heiko Hamann, and Gregor Engels. From user demand to software service: using machine learning to automate the requirements specification process. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 379–385. IEEE, 2017.

[39] N. Englisch, A. Heller, and W. Hardt. Automated generation of tests for education in software engineering. In *EDULEARN18 Proceedings*, 10th International Conference on Education and New Learning Technologies, pages 8478–8484. IATED, 2-4 July, 2018 2018.

[40] Norbert Englisch, Roland Mittag, Felix Hänchen, Owes Khan, Alejandro Masrur, and Wolfram Hardt. Efficiently testing autosar software based on an automatically generated knowledge base. In *Simulation and Testing for Vehicle Technology*, pages 87–97. Springer, 2016.

[41] Norbert Englisch, Felix Hänchen, Frank Ullmann, Alejandro Masrur, and Wolfram Hardt. Application-driven evaluation of autosar basic software on modern ecus. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 60–67. IEEE, 2015.

[42] Ventsislav Yordanov. Introduction to natural language processing for text. `https://towardsdatascience.com/introduction-to-natural-language-processing-for-text-df845750fb63/`, 2018.

[43] Mariana Neves. Introduction to Part-of-Speech Tagging. `https://hpi.de/fileadmin/user_upload/fachgebiete/plattner/teaching/NaturalLanguageProcessing/NLP2016/NLP04_PartOfSpeechTagging.pdf/`, 2016.

[44] Adwait Ratnaparkhi. *Maximum Entropy Models for Natural Language Ambiguity Resolution*. PhD thesis, Philadelphia, PA, USA, 1998. AAI9840230.

[45] Christopher Manning. Neural Network Dependency Parser. `https://nlp.stanford.edu/software/nndep.html`, 2014.

[46] SM Rajkumar. Equivalence Partitioning Test Case Design Technique. `https://www.softwaretestingmaterial.com/equivalence-partitioning-testing-technique/`, 2018.

[47] Benjamin L Kovitz. *Practical software requirements: a manual of content and style*. Manning Publications Co., 1998.

[48] Karl Eugene Wiegers. *Creating a software engineering culture*. Pearson Education, 1996.

[49] Davis Alan. Software requirements: Objects, functions, and states. *Englewood Cliffs*, 1993.

[50] Software Engineering Standards Committee et al. Ieee guide for information technology–system definition–concept of operations (conops) document. *IEEE Std*, pages 1362–1998, 1998.

[51] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[52] Gregory D. Githens. Customer centered products: Creating successful products through smart requirements management: Ivy f. hooks and kristin a. farry;new york: Amacon, 2000. *Journal of Product Innovation Management*, 18:350351, 09 2001.

[53] Ian F. Alexander and Richard Stevens. *Writing Better Requirements*. Pearson Education, 2002.

[54] Andrew P. Sage and William B. Rouse. *Handbook of systems engineering and management*. 1999.

[55] Microsoft Coorperation. What is MS Excel. `https://www.greycampus.com/opencampus/ms-excel/what-is-ms-excel/`, 2013.

[56] Steven Bird, Ewan Klein, and Edward Loper. Natural Language Processing with Python. `https://www.nltk.org/book/`, 2019.

[57] Dekang Lin. On the structural complexity of natural language sentences. In *COLING 1996 Volume 2: The 16th International Conference on Computational Linguistics*, 1996.

[58] Muneera Bano. Addressing the challenges of requirements ambiguity: A review of empirical literature. In *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*, pages 21–24. IEEE, 2015.

A. Appendix

A.1. Some Bad Requirement Examples

A.1.1. Example 1

"The Fuel tank sensor shall provide status messages at continuous interval not less than every 40 seconds"

Remarks

- Under which scenario and in which manner it will be sent to user?
- What will be the visibility duration of status messages?
- The time interval is not obvious and word "every" is ambiguous
- The upper bound is not define.

A.1.2. Example 2

"The system shall produce door state error report that permits instant resolution of errors when utilized by Json invoices"

Remarks

- The word "instant" indicates an activity performed by an individual.
- No description of error report shows that this requirement is incomplete.
- It is also not known when the report is generated.
- How this requirement would be verified is also not clearly stated?

A.1.3. Example 3

"The system shall not provide input and search option that could be undesirable"

Remarks

- The term "undesirable" is ambiguous.
- The above give requirement is an inverse requirement statement. Therefore it is always advised to avoid the use of inverse requirements which states what the system will not do.